Rozpoczęcie pracy w programie Microsoft Transaction Server

Program Microsoft® Transaction Server (MTS) jest opartym na składnikach systemem przetwarzania transakcji, pozwalającym projektować, rozmieszczać i zarządzać skalowalnymi, niezawodnymi aplikacjami o wysokiej wydajności, przeznaczonymi do pracy w sieciach Internet i intranet. Program MTS definiuje model programistyczny dla projektowania rozpowszechnianych aplikacji opartych na składnikach. Ponadto oferuje środowisko czasu wykonywania oraz narzędzia graficzne do rozmieszczania tworzonych aplikacji i zarządzania nimi.

Niniejszy rozdział zawiera zwięzły przegląd nowości wprowadzonych do programu MTS, krótkie omówienie dokumentacji oraz słownik użytecznych terminów. W rozdziale omówiono następujące tematy:

- Co nowego w programie MTS
- Omówienie dokumentacji programu MTS
- Słownik programu MTS
- Narzędzia programu MTS
- Często zadawane pytania na temat programu MTS

Co nowego w programie MTS

Program Microsoft Transaction Server (MTS) w wersji 2.0 oferuje szereg nowych funkcji, ułatwiających rozmieszczanie skalowalnych, niezawodnych aplikacji, przeznaczonych do pracy w sieciach Internet i intranet. W niniejszym podrozdziale opisano nowe możliwości programu MTS.

Pełna integracja z programem Internet Information Server (IIS) w wersji 4.0

Program MTS 2.0 jest ściśle zintegrowany z programem IIS 4.0, dzięki czemu można go uznać za najlepszą platformę programistyczną dla aplikacji biznesowych działających w sieci Web. Do nowych funkcji związanych z integracją programów MTS i IIS należą:

- Środowisko transakcyjne Active Server Pages

  W środowisku Active Server Pages skrypty mogą być wykonywane w ramach transakcji zarządzanych przez program MTS. Dzięki temu mechanizmy ochrony transakcji MTS mogą być stosowane do całej aplikacji (działającej w sieci Web).
- Zabezpieczenia na wypadek awarii aplikacji IIS

  Aplikacje IIS dla sieci Web mogą być wykonywane we własnych pakietach programu MTS, dzięki czemu mogą one korzystać z mechanizmu izolowania procesów i ochrony na wypadek awarii.
- Zdarzenia transakcyjne

  Projektanci mogą osadzać polecenia w skryptach w środowisku Active Server Pages, dzięki czemu możliwe staje się dostosowywanie odpowiedzi aplikacji działających w sieci Web w zależności od wyniku transakcji.
- Kontekst obiektowy dla obiektów wbudowanych programu IIS

  Śledzenie informacji o stanie użytkowników, kontrolowane dotychczas przez obiekty wbudowane programu IIS, jest teraz obsługiwane przez mechanizm kontekstu obiektowego programu MTS. Dzięki temu projektanci aplikacji dla sieci Web mogą korzystać z prostego modelu programistycznego MTS.
- Wspólna instalacja i zarządzanie

  Programy MTS i IIS korzystają ze wspólnej instalacji i wspólnej konsoli zarządzania, dzięki czemu znacznie mniej złożone staje się rozmieszczanie i zarządzanie aplikacjami biznesowymi w sieci Web.

Obsługa protokołu transakcyjnego XA, w tym obsługa baz danych Oracle

- Program MTS 2.0 obsługuje protokół transakcyjny XA, który umożliwia współpracę aplikacji MTS z bazami danych typu IBM DB2, Informix i innymi zgodnymi z protokołem XA, działającymi w systemie operacyjnym Windows NT Server, a także w innych systemach opracowanych poza firmą Microsoft (na przykład w różnych wersjach systemu UNIX) przy użyciu systemu ODBC.
- Sprzedawcy baz danych zgodnych z protokołem XA mają możliwość testowania swoich sterowników ODBC pod kątem ich przyszłej współpracy z programem MTS 2.0; Więcej informacji na ten temat można uzyskać bezpośrednio od sprzedawcy danej bazy danych.
- Program MTS 2.0 zawiera uaktualniony sterownik ODBC firmy Microsoft dla baz danych Oracle, który pozwala aplikacjom MTS na bezpośrednie zarządzanie transakcjami z poziomu programu Oracle w wersji 7.3 lub nowszej przy użyciu systemu ODBC.

Obsługa systemów operacyjnych typu desktop

Program MTS 2.0 może współpracować zarówno z systemem operacyjnym Windows NT wersja 4.0, jak i Windows 95. Zapewniana przez program MTS obsługa systemów operacyjnych typu desktop firmy Microsoft pozwala tworzyć i rozmieszczać autonomiczne wersje aplikacji MTS.

- System Windows 95 może być wykorzystywany jako platforma projektowa dla opracowywania składników MTS, rozmieszczanych później za pomocą serwera wyposażonego w system Windows NT. Komputery wyposażone w system Windows 95 mogą być klientami administracyjnymi dla aplikacji serwerów MTS działających w systemie Windows NT. Ponadto program MTS oferuje

środowisko czasu wykonywania dla programu Personal Web Server (PWS), który działa w systemie Windows 95.

- Ze względu na różnice między systemami Windows NT i Windows 95, w systemie Windows 95 program MTS nie może współpracować z programem Microsoft Cluster Server oraz nie obsługuje zabezpieczeń opartych na rolach. Komputery wyposażone w system Windows 95 (korzystające z programu MTS) nie mogą być zdalnie administrowane ani za pomocą komputerów korzystających z systemu Windows NT, ani za pomocą komputerów wyposażonych w system Windows 95. Ponadto, w systemie Windows 95 nie są instalowane przykładowe skrypty (napisane w języku programowania Microsoft Visual Basic Scripting Edition (VBScript)), ilustrujące metody automatyzacji administrowania w programie MTS.

Obsługa programu Microsoft Cluster Server

Program MTS 2.0 współpracuje z programem Microsoft Cluster Server (MSCS), który umożliwia automatyczną pracę bezawaryjną pakietów MTS w klastrze. Dzięki automatycznej opcji pracy bezawaryjnej aplikacje MTS zyskują na dostępności.

Obsługa transakcji CICS i IMS za pośrednictwem LU 6.2 Sync Level 2

Program MTS 2.0 pozwala rozmieszczać aplikacje MTS obsługujące transakcje CICS i IMS w środowisku MVS. Program MTS oferuje pełną obsługę wersji beta 2 lub nowszej programu Cedar, składnika programu SNA Server, który zapewnia współpracę między programami MTS, CICS/MVS i IMS/MVS.

Nowe mechanizmy administrowania

Program MTS 2.0 oferuje wiele nowych mechanizmów administrowania, istotnie ułatwiających rozmieszczanie pakietów MTS i administrowanie nimi. Należą do nich:

- Przystawki programu MTS Explorer

  Program MTS Explorer jest teraz przystawką programu Microsoft Management Console (MMC). Pakietami MTS można zarządzać z tej samej konsoli administracyjnej, która obsługuje inne produkty, na przykład program IIS. Więcej informacji na temat programu MMC oraz korzystania z przystawek można uzyskać w dokumentacji programu MMC.

- Zamykanie pojedynczych procesów serwera

  W programie MTS 2.0 możliwe jest zamykanie pojedynczych pakietów (za pomocą programu MTS Explorer)   bez konieczności zamykania procesu serwera programu MTS. Zamknięcie pakietu powoduje zakończenie procesu serwera aplikacji.

- Udoskonalone ustawienia aktywacji pakietów

  W programie MTS 2.0 aktywacja jest ustawiana wyłącznie na poziomie pakietu. Zrezygnowano z mechanizmu mieszanej aktywacji dla różnych składników pakietu. Co więcej, pakiety nie mogą działać z ustawieniem aktywacji zdalnej. Aktywację można ustawić albo jako **Serwer** (lokalna aktywacja składników) albo jako **Biblioteka** (aktywacja składników powodowana przez proces klienta).

  Pakiety utworzone w programie MTS 1.0 zawierające wyłącznie składniki wewnątrzprocesowe po zainstalowaniu programu MTS 2.0 są automatycznie uaktualniane i traktowane jako pakiety bibliotek. Pakiety utworzone w programie MTS 1.0 zawierające wyłącznie składniki ustawione do aktywacji lokalnej są uaktualniane i traktowane jako pakiety serwera. Pakiety utworzone w programie MTS 1.0 zawierające składniki mieszane należy ręcznie skonfigurować jako pakiety bibliotek lub serwera.

- Możliwość wyboru wielu elementów w celu uaktualnienia właściwości

  Za pomocą programu MTS Explorer można zaznaczyć i modyfikować właściwości wielu elementów jednocześnie.

- Przenoszenie składników między pakietami

  Składniki można przenosić między pakietami, przeciągając je z jednego pakietu i upuszczając w drugim.

Nowe mechanizmy programowania

Dzięki nowym mechanizmom programowania w programie MTS 2.0 o wiele łatwiej jest budować aplikacje MTS. Wśród nowości tego typu występują:

- Nowe i uaktualnione przykładowe aplikacje MTS

  Oprócz uaktualnionej aplikacji Sample Bank program MTS oferuje dwie nowe aplikacje przykładowe. Aplikacja Tic-Tac-Toe jest prostą grą dla wielu użytkowników i obrazuje metody współużytkowanego zarządzania składnikami nietransakcyjnymi. Przykładowe skrypty administracyjne pokazują, w jaki sposób za pomocą skryptowych obiektów administracyjnych można zautomatyzować niektóre procedury programu MTS Explorer (przy użyciu skryptów Windows Scripting Host).
- Skryptowe obiekty administracyjne

  Przy użyciu skryptowych obiektów administracyjnych można automatyzować wykonywane w programie MTS Explorer procedury rozmieszczania i utrzymywania żądanego stanu pakietów. Za pomocą dowolnego języka programowania zgodnego z Automatyzacją można pisać proste skrypty automatyzujące takie czynności, jak instalacja pakietów.
- Dokumentacja związana z projektowaniem i implementacją aplikacji MTS

  W podręczniku *MTS Programmer's Guide* opisano różne techniki projektowania i implementacji, wykorzystywane podczas tworzenia aplikacji MTS. W książce omówiono wiele zagadnień: od osadzania w składnikach reguł logiki biznesu do problemów diagnozowania i debugowania.

Zobacz też

Wprowadzenie do podręcznika Administrator's Guide, Przegląd informacji i pojęć dotyczących programu MTS

Omówienie dokumentacji programu MTS

Program Microsoft Transaction Server (MTS) zawiera bogatą dokumentację, pomocną w projektowaniu, budowaniu, rozmieszczaniu i administrowaniu aplikacjami MTS.

Książka/Rozdział	Opis
Instalacja programu MTS	W rozdziale opisano sposób instalacji programu MTS i jego składników, a także omówiono kwestie dostępu aplikacji MTS do baz danych Oracle oraz instalowania przykładowych aplikacji MTS.
Rozpoczęcie pracy w programie MTS	W rozdziale dokonano przeglądu nowości wprowadzonych w programie MTS, opisano dokumentację programu oraz podano słownik używanej terminologii.
Krótkie wprowadzenie do programu MTS	Rozdział zawiera przegląd informacji na temat programu MTS.
MTS Administrator's Guide	
Wprowadzenie do podręcznika MTS Administrator's Guide	W rozdziale opisano różne metody wykorzystania programu MTS Explorer do rozmieszczania aplikacji MTS i administrowania nimi oraz zestawiono informacje o interfejsie graficznym programu.
Tworzenie pakietów MTS	Rozdział zawiera dokumentację związaną z konkretnymi zadaniami tworzenia i składania pakietów MTS.
Rozpowszechnianie pakietów MTS	Rozdział zawiera dokumentację związaną z konkretnymi zadaniami rozpowszechniania pakietów MTS.
Instalacja pakietów MTS	Rozdział zawiera dokumentację związaną z konkretnymi zadaniami instalowania i konfigurowania pakietów MTS.
Utrzymywanie żądanego stanu pakietów MTS	W rozdziale omówiono zagadnienia utrzymywania żądanego stanu i monitorowania pakietów MTS (również na przykładzie konkretnych zadań).
Zarządzanie transakcjami MTS	W rozdziale omówiono rozpowszechnianie transakcji oraz zarządzanie transakcjami za pomocą programu MTS Explorer.
Automatyzacja administrowania w programie MTS	Rozdział zawiera przegląd pojęć, procedury oraz przykładowe kody programów, objaśniające zastosowanie obiektów skryptowych MTS do automatyzacji różnych czynności programu MTS Explorer.

MTS Programmer's Guide

Overview and Concepts	W rozdziale scharakteryzowano program MTS i istotę współpracy jego składników, przedstawiono zalety programu przy projektowaniu aplikacji typu "klient/serwer" i administrowaniu systemem oraz omówiono szczegółowo zagadnienia związane z programowaniem składników MTS.
Building Applications for MTS	Zawiera informacje o projektowaniu składników ActiveX™ dla programu MTS (dotyczące konkretnych zadań).
MTS Administrative Reference	Zawiera informacje na temat wykorzystania obiektów skryptowych programu MTS do automatyzowania wybranych czynności programu MTS Explorer.
MTS Reference	Zawiera informacje o korzystaniu z interfejsu programowania aplikacji API (z ang. "application programming interface").
MTS Administrative Reference	Zawiera informacje na temat wykorzystania obiektów skryptowych programu MTS do automatyzowania wybranych czynności programu MTS Explorer.

Narzędzia programu MTS

Program MTS oferuje szereg narzędzi wiersza poleceń, które można wykorzystać do automatyzacji pewnych zadań, na przykład za pomocą plików wsadowych (narzędzia te są dostępne wprost z wiersza poleceń).

Narzędzia wiersza poleceń

W poniższej tabeli zestawiono najważniejsze informacje na temat narzędzi wiersza poleceń instalowanych razem z programem MTS.

Narzędzie	Funkcja
MTXSTOP.exe	Powoduje zamknięcie wszystkich procesów programu MTS. Narzędzie stanowi odpowiednik opcji **Zamknij procesy serwera**, dostępnej po kliknięciu prawym przyciskiem myszy w oknie **Mój komputer**.
MTXTEST.exe	Pozwala przetestować kod przekazywania składnika na zewnątrz środowiska czasu wykonywania programu MTS.
MTXTSTOP.exe	Powoduje zatrzymanie narzędzia MTXTEST.exe. Narzędzie to jest instalowane tylko w ramach instalacji projektowej.
SAMPDTCC.exe	Pozwala przetestować instalację usługi MS DTC z przykładowym klientem.
SAMPDTCS.exe	Pozwala przetestować instalację usługi MS DTC z przykładowym serwerem.
MTXREREG.exe	Powoduje odświeżenie wszystkich składników zarejestrowanych na komputerze. Narzędzie stanowi odpowiednik opcji **Odśwież składniki**, dostępnej po kliknięciu prawym przyciskiem myszy w ramach zaznaczonego pakietu.
MTXREPL.exe	Pozwala zreplikować serwer MTS. Obydwa komputery, źródłowy i docelowy, muszą być uruchomione.
TestOracleXAConfig.Exe	Pozwala przetestować konfigurację bazy danych Oracle w celu sprawdzenia poprawności rozpowszechnianych transakcji zawierających składniki MTS. Jeśli narzędzie daje wynik negatywny, oznacza to, że rozpowszechniane transakcje nie mogą współpracować z bazami danych Oracle.

Narzędzia administracyjne systemu Windows NT

System Windows NT oferuje szereg narzędzi pomocnych przy administrowaniu aplikacjami MTS. Aby skorzystać z tych narzędzi, należy kliknąć przycisk **Start**, wskazać polecenie **Programy**, a następnie menu **Narzędzia administracyjne (Wspólne)**.

Narzędzie		Funkcja
	Przeglądarka zdarzeń	W systemie Windows NT zdarzenie oznacza pewien istotny fragment działań systemu lub programu, o którym użytkownik powinien być powiadomiony. Przeglądarka zdarzeń albo

		powiadamia użytkownika, albo umieszcza zdarzenie w dzienniku. W razie jakichkolwiek problemów z aplikacją MTS w pierwszym rzędzie należy skorzystać z Przeglądarki zdarzeń.
	Monitor wydajności	Monitor wydajności jest narzędziem służącym do monitorownia wydajności danego komputera lub innych komputerów w sieci.
	Menedżer serwerów	Menedżer serwera umożliwia wyświetlenie listy stacji roboczych i serwerów działających w danej domenia.
	Menedżer użytkowników dla domen	Menedżer użytkowników dla domen pozwala zakładać, usuwać i wyłączać konta użytkowników domeny. Za pomocą tego narzędzia można również dodawać konta użytkowników do grup i określać ustawienia zabezpieczeń.
	Diagnostyka systemu Windows NT	Diagnostyka systemu Windows NT pozwala wyświetlić informacje o zasobach używanego komputera.

Uwaga Wiele z wymienionych wyżej narzędzi administracyjnych systemu Windows NT wymaga, aby korzystający z nich użytkownik był zalogowany na komputerze wyposażonym w uprawnienia administracyjne.

Zobacz też

Wprowadzenie do podręcznika MTS Administrator's Guide

Często zadawane pytania na temat programu MTS

Często zadawane pytania na temat pracy w programie Microsoft Transaction Server są zlokalizowane pod adresem: http://www.microsoft.com/support/transaction/content/faq/.

Krótkie wprowadzenie do programu Microsoft Transaction Server

Program Microsoft® Transaction Server (MTS) jest opartym na składnikach systemem przetwarzania transakcji, pozwalającym tworzyć, rozmieszczać i administrować skalowalnymi, niezawodnymi aplikacjami o wysokiej wydajności, przeznaczonymi do pracy w sieciach Internet i intranet.

Poniższe podrozdziały zawierają informacje na temat różnych funkcji i elementów programu Microsoft Transaction Server:

- Do czego służy program Microsoft Transaction Server?
- Środowisko czasu wykonywania programu Microsoft Transaction Server
- Program Microsoft Transaction Server Explorer
- Interfejsy API w programie Microsoft Transaction Server
- Przykładowe aplikacje programu Microsoft Transaction Server

Do czego służy program Microsoft Transaction Server?

Program MTS jest opartym na składnikach systemem przetwarzania transakcji, pozwalającym projektować, rozmieszczać i zarządzać złożonymi aplikacjami serwerowymi, przeznaczonymi do pracy w sieciach Internet i intranet. Ponadto, program MTS oferuje projektantom zaawansowane narzędzie graficzne (MTS Explorer), służące do rozmieszczania i administrowania serwerowymi aplikacjami MTS.

Program MTS udostępnia następujące funkcje i narzędzia:‹

- Środowisko czasu wykonywania programu MTS.
- Program MTS Explorer, graficzny interfejs użytkownika służący do rozmieszczania składników aplikacji i zarządzania nimi.
- Interfejsy programowania aplikacji oraz *rozdzielacz zasobów,* pozwalające czynić aplikacje skalowalnymi i niezawodnymi. Rozdzielacze zasobów są to specjalne usługi zarządzające w imieniu składników aplikacji nietrwałymi stanami współużytkowania zasobów w ramach danego procesu.
- Trzy przykładowe aplikacje obrazujące, po pierwsze metody budowania składników MTS za pomocą interfejsu programowania aplikacji, po drugie różne możliwości wykorzystania skryptowych obiektów administracyjnych do automatyzacji procedur rozmieszczania w programie MTS Explorer.

Model programistyczny programu MTS umożliwia projektowanie składników realizujących różne funkcje biznesowe. Środowisko czasu wykonywania programu MTS pozwala uruchamiać te składniki. Za pomocą programu MTS Explorer można z kolei zarządzać składnikami uruchomionymi w środowisku czasu wykonywania, jak również rejestrować je.

Trójstopniowy model programistyczny pozwala projektantom i administratorom wykraczać poza ograniczenia dwustopniowych aplikacji typu klient/serwer. Można wyróżnić dwie ważne zalety modelu trójstopniowego, ułatwiające rozmieszczanie aplikacji i zarządzanie nimi:

- W modelu trójstopniowym istotna jest raczej logiczna struktura aplikacji, a nie jej kształt fizyczny. Każda z usług może rezydować gdziekolwiek i wywoływać dowolne inne usługi.
- Aplikacje są rozpowszechnione, a zatem odpowiednie składniki mogą być uruchaminae oddzielnie, we właściwych miejscach. Dzięki temu użytkownicy mogą w bardziej optymalny sposób korzystać z sieci i zasobów komputera.

Zobacz też

Omówienie dokumentacji programu MTS

Środowisko czasu wykonywania programu Microsoft Transaction Server

Środowisko czasu wykonywania programu MTS ułatwia projektowanie, rozmieszczanie i zarządzanie aplikacją, oferując projektantom i administratorom systemu bogaty (chociaż łatwy w użyciu) zestaw usług systemowych. Zestaw ten obejmuje między innymi:

- Rozpowszechniane transakcje. *Transakcja* jest fragmentem działań realizowanych jako operacja atomowa ľ to znaczy wykonywana lub nie wykonywana w całości.
- Automatyczne zarządzanie procesami i wątkami.
- Zarządzanie instancjami obiektów.
- Rozproszony serwis zabezpieczeń, umożliwiający kontrolę nad tworzeniem i wykorzystywaniem obiektów.
- Interfejs graficzny, pozwalający administrować systemem i zarządzać składnikami.

Dzięki powyższym usługom systemowym projektanci mogą po pierwsze opracowywać aplikacje skalowalne i niezawodne, po drugie skupić się raczej na rozwiązywaniu konkretnych problemów, niż na obmyślaniu infrastruktury systemu.

Program MTS oferuje możliwość współpracy z dowolnymi narzędziami do projektowania aplikacji, pod warunkiem, że obsługują one tworzenie bibliotek DLL w technologii ActiveX. Na przykład projektując aplikacje można korzystać z programów Microsoft Visual Basic, Microsoft Visual C++, Microsoft Visual J++ oraz dowolnych innych narzędzi zgodnych z technologią ActiveX.

Program MTS może współpracować z wieloma programami do obsługi zasobów, między innymi z systemami relacyjnych baz danych, systemami plików i systemami przechowywania dokumentów. Dzięki temu projektanci i niezależne firmy projektowe mogą dostosować projektowane aplikacje do współpracy z jednym lub dwoma programami do obsługi zasobów, zachowując jednocześnie wszystkie korzyści płynące ze stosowania transakcji (lokalnych lub rozpowszechnianych).

Zobacz też

MTS Overview and Concepts

Program Microsoft Transaction Server Explorer

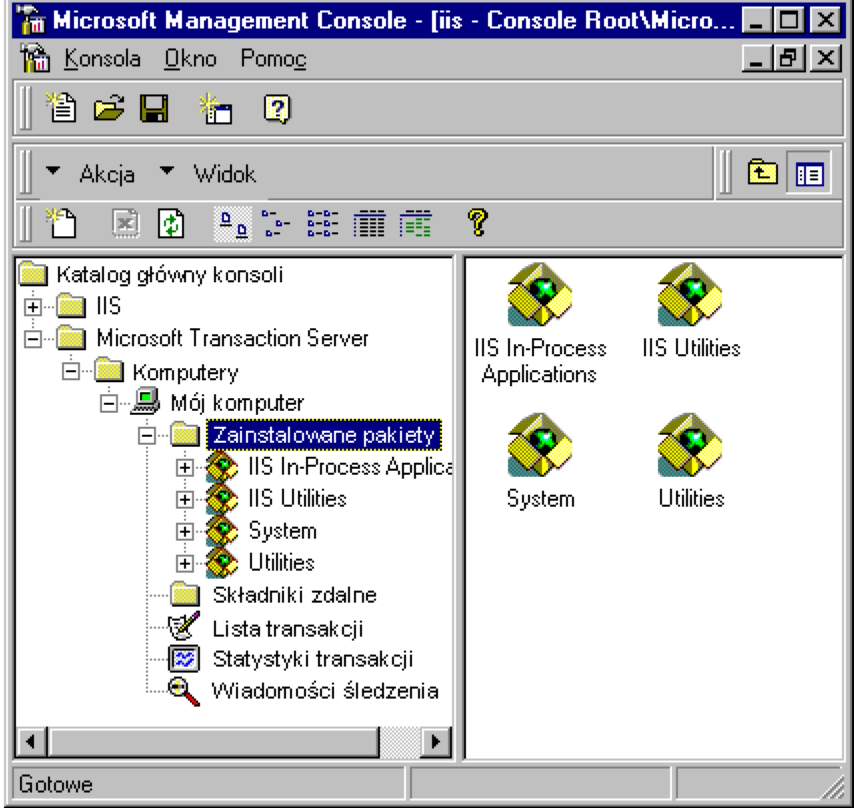
Program MTS Explorer jest graficznym interfejsem użytkownika, służącym do rozmieszczania składników MTS i zarządzania nimi. Program jest przeznaczony dla projektantów aplikacji, administratorów systemu i administratorów sieci Web. Za jego pomocą mogą oni administrować, rozpowszechniać, instalować, rozmieszczać i testować pakiety. Projektanci wykorzystują często program MTS Explorer do łączenia składników w pakietach wstępnie wbudowanych, a także do rozpowszechniania i testowania składników w środowisku programu MTS. Administratorzy i projektanci używają programu MTS Explorer do instalacji, rozmieszczania i utrzymywania żądanego stanu składników pakietów. Ponadto program oferuje narzędzia do monitorowania i zarządzania transakcjami dla składników.

Hierarchia programu MTS Explorer ukazuje obiekty obsługiwane przez program oraz sposób ich uporządkowania w środowisku czasu wykonywania. Do obiektów tych należą:

- Komputery
- Pakiety
- Składniki
- Role
- Interfejsy
- Metody

Pakiety MTS są instalowane na komputerach, zawierają zaś składniki, a definiują role. Składniki pakietów definiują interfejsy i metody. Program MTS Explorer daje możliwość korzystania z wyspecjalizowanych okien do przeglądania informacji dotyczących transakcji i komunikatów śledzenia.

Poniższy diagram ukazuje sposób wyświetlania hierarchii w lewym okienku programu MTS Explorer:

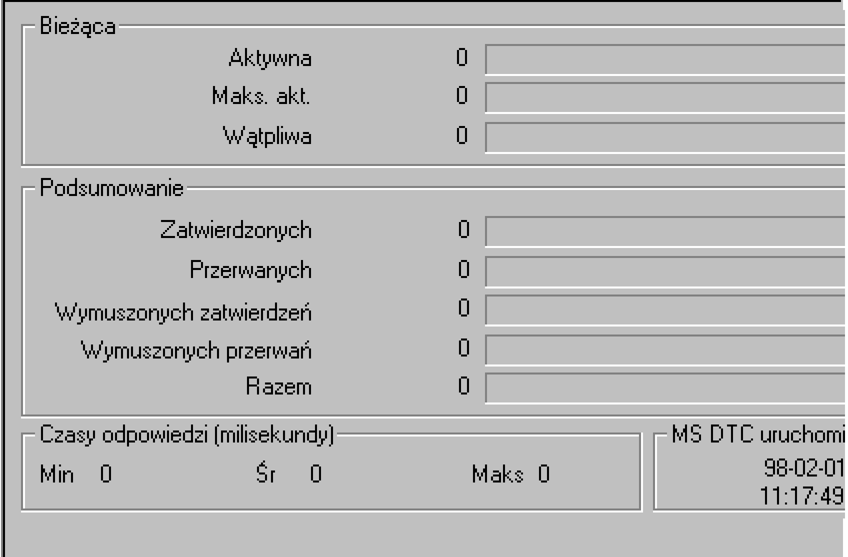


Za pomocą okien właściwości można przeglądać właściwości składników pakietu (zob. poniższy

diagram) oraz pakiety zainstalowane na komputerze.

Prog ID	Transakcja	DLL	CLSID	Wątkowość	Zabezpiecz
Bank. Account	Wymagana	C:\Progra..	{5BE6C9DB...	Apartament	Y
Bank. Account. VC	Wymagana	C:\Progra..	{04CF0B76-...	Oba	Y
Bank. Account. VJ	Wymagana	C:\Progra..	{9FAF8612-...	Oba	Y
Bank. CreateTable	Wymagana nowa	C:\Progra..	{5BE6C9E1-...	Apartament	Y
Bank. GetReceipt	Obsługiwana	C:\Progra..	{5BE6C9DF...	Apartament	Y
Bank. GetReceipt. VC	Wymagana nowa	C:\Progra..	{A81260B2-...	Oba	Y
Bank. MoveMoney	Wymagana	C:\Progra..	{5BE6C9DD...	Apartament	Y
Bank. MoveMoney. VC	Wymagana	C:\Progra..	{04CF0B7B-...	Oba	Y
Bank. UpdateReceipt	Wymagana nowa	C:\Progra..	{5BE6C9E3-...	Apartament	Y
Bank. UpdateReceipt. VC	Wymagana nowa	C:\Progra..	{A81260B8-...	Oba	Y

Korzystając z okna **Statystyki transakcji** można przeglądać zbiorcze statystyki ostatnich transakcji.



Przeglądanie hierarchii programu MTS Explorer

Okno programu MTS Explorer zawiera dwa okienka: w lewym jest wyświetlana hierarchia obiektów, w prawym zaś zawartość obiektu klikniętego w lewym okienku. Hierarchia ma strukturę drzewiastą, a zawiera foldery i wszystkie obiekty konfigurowane za pomocą programu MTS Explorer.

Aby obejrzeć zawartość danego folderu lub innego elementu hierarchii programu MTS Explorer, można klinąć go dwukrotnie w prawym okienku programu. Można również kliknąć go w lewym okienku; jego zawartość zostanie wówczas wyświetlona w prawym okienku. Aby rozwinąć dany element hierarchii, należy kliknąć widoczny obok znak (+); hierarchia tego elementu zostanie wyświetlona w lewym okienku. Kliknięcie dwukrotne folderu lub innego elementu w prawym okienku powoduje wyświetlenie jego zawartości w prawym okienku, dzięki czemu można przełączać się między zwiniętymi i rozwiniętymi postaciami hierarchii.

W obu okienkach do przechodzenia między różnymi elementami (zaznaczania ich) służą klawisze strzałek. Naciśnięcie klawisza ENTER powoduje wyświetlenie zawartości zaznaczonego elementu. Klawisz TAB umożliwia przełączanie się między dwoma okienkami programu.

W systemie Windows 95 w lewym okienku programu MTS Explorer nie jest wyświetlane drzewo hierarchii. Aby przeglądać hierarchię w dół, należy klikać dwukrotnie ikony elementów, aby przeglądać hierarchię w górę, należy używać przycisku **W górę o jeden poziom** z paska narzędzi.

Więcej informacji na temat korzystania z programu MTS Explorer w systemie Windows 95 można uzyskać pod hasłem Wprowadzenie do podręcznika MTS Administrator's Guide.

Ustawianie i przeglądanie właściwości elementów

Podstawowe informacje o elementach dodawanych do hierarchii programu MTS Explorer są wyświetlane na ich arkuszach właściwości. Rodzaj informacji zależy od elementu. Na przykład arkusz właściwości dla komputera zawiera nazwę komputera, lokalizację pliku dziennika oraz ustawienia aktualizacji, podczas gdy arkusz dla pakietu zawiera informacje o zabezpieczeniu i innych ustawieniach dotyczących procesów.

Aby wyświetlić arkusz właściwości można zaznaczyć element i wybrać polecenie **Właściwości** z menu **Akcja** lub kliknąć element prawym przyciskiem myszy i wybrać polecenie **Właściwości** z menu podręcznego. Każdy arkusz składa się z pojedynczych stron, które można przeglądać, klikając odpowiadające im karty.

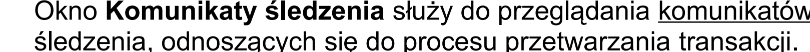
Monitorowanie i przeprowadzanie transakcji

Najważniejszą funkcją środowiska czasu wykonywania programu MTS jest obsługa przetwarzania transakcji. Służy do tego wyspecjalizowana usługa Microsoft Distributed Transaction Coordinator (MS DTC). Usługa MS DTC stanowi element systemu Windows NT, a program MTS korzysta z niej przed sfinalizowaniem transakcji, w celu sprawdzenia, czy wszystkie strony transakcji osiągnęły porozumienie.

Program MTS Explorer oferuje trzy okna hierarchiczne do obsługi transakcji:

Okno **Lista transakcji** służy do monitorowania stanu transakcji aktywnej.

Okno **Statystyki transakcji** służy do przeglądania informacji o zbiorczych statystykach ostatnich transakcji.

 Okno **Komunikaty śledzenia** służy do przeglądania komunikatów śledzenia, odnoszących się do procesu przetwarzania transakcji.

Zobacz też

Wprowadzenie do podręcznika MTS Administrator's Guide

Interfejsy API programu Microsoft Transaction Server

Za pomocą interfejsów programowania aplikacji (interfejsów API) programu MTS można projektować niezawodne i skalowalne aplikacje MTS, wykorzystujące w pełni różne funkcje środowiska czasu wykonywania programu MTS, a także automatyzować procedury administrowania pakietami i składnikami.

Projektowanie aplikacji klientów

W aplikacjach klienckich, uruchamianych na zewnątrz środowiska czasu wykonywania programu MTS, instancje obiektów programu MTS są tworzone za pomocą standardowych funkcji bibliotek COM (funkcji **CoCreateInstance** w języku C++; metody **CreateObject** w języku Visual Basic).

Projektowanie składników

Projektując składniki MTS (składniki serwerowe, które będą rejestrowane w środowisku czasu wykonywania programu MTS), można korzystać z interfejsów **IObjectContext**, **ISharedPropertyGroupManager**, **ISharedPropertyGroup** i **ISharedProperty**. Interfejsy te służą do:

- Deklarowania, że praca danego obiektu została zakończona
- Uniemożliwienia przekazu transakcji
- Tworzenia obiektów MTS
- Włączania prac obiektów w zakres transakcji bieżącego obiektu
- Sprawdzania, czy wywołujący zawiera się w konkretnej roli
- Sprawdzania, czy włączono zabezpieczenie

Automatyzacja administrowania w programie MTS

Do automatyzacji procesu administrowania składnikami i pakietami służą odpowiednie obiekty administracyjne programu MTS. Obiektów tych można używać za pośrednictwem języków programowania Visual Basic, Visual Basic Scripting Edition (VBScript) i dowolnych innych, zgodnych z Automatyzacją. Za ich pomocą można automatyzować najróżniejsze procedury programu MTS Explorer: od instalacji pakietów wstępnie wbudowanych do wyliczania powiązanych ze sobą kolekcji.

Zobacz też

Wprowadzenie do podręcznika MTS Administrator's Guide, MTS Overview and Concepts,MTS Reference, MTS Administrative Reference

Przykładowe aplikacje programu Microsoft Transaction Server

Oprócz bogatej dokumentacji program MTS oferuje projektantom przykładowe aplikacje, które mogą być użyteczne w poznawaniu różnych funkcji programu. Ich fragmenty można kopiować do własnych aplikacji, a jeśli to niezbędne dostosowywać do własnych potrzeb.

W książce MTS Programmer's Guide omówiono wiele przykładowych kodów programów oraz aplikacji, obrazujących różne techniki programowania w środowisku MTS. Wiele plików dla tych aplikacji stanowi element instalacji programu. Są one przechowywane w folderze \Samples używanej instalacji programu MTS.

Program MTS oferuje następujące aplikacje przykładowe.

Aplikacja	Opis
Sample Bank	Aplikacja Sample Bank jest prostym programem do obsługi transakcyjnej bazy danych, pokazującym, w jaki sposób należy korzystać z interfejsów programowania aplikacji programu MTS
Tic-Tac-Toe	Aplikacja Tic-Tac-Toe jest prostą grą dla wielu użytkowników, prezentującą zastosowania składników nietransakcyjnych do zarządzania stanem współużytkowania.
Przykładowe skrypty administracyjne	Skrypty obiektów administracyjnych pokazują, w jaki sposób za pomocą języka VBScript można automatyzować procedury programu MTS Explorer.

Zobacz też

Instalacja dokumentacji i przykładów projektowania w programie MTS, Instalacja przykładowej aplikacji Sample Bank, Instalacja przykładowej aplikacji Tic-Tac-Toe, Instalacja przykładowych skryptów administracyjnych programu MTS, MTS Overview and Concepts,Przykład skryptu utworzonego w programie Visual Basic, automatyzującego administrowanie w programie MTS

Instalacja programu Microsoft Transaction Server

Zapraszamy do pracy w programie Microsoft® Transaction Server (MTS), który pozwala w łatwy sposób projektować i rozmieszczać niezawodne, skalowalne aplikacje o wysokiej wydajności, przeznaczone do pracy w sieciach Internet i intranet. Program MTS definiuje model programistyczny dla projektowania rozpowszechnianych aplikacji opartych na składnikach. Ponadto oferuje środowisko czasu wykonywania oraz narzędzia graficzne do rozmieszczania tworzonych aplikacji i zarządzania nimi.

Informacje na temat instalacji i konfiguracji programu MTS zebrano w następujących podrozdziałach:

- Wymagania systemowe programu MTS
- Instalacja dokumentacji i przykładów projektowania w programie MTS
- Konfiguracja serwera programu MTS
- Konfiguracja programu MTS z programem Microsoft Cluster Server
- Ustawianie programu MTS dla potrzeb współpracy z programem Oracle
- Instalacja przykładowej aplikacji Sample Bank
- Instalacja przykładowej aplikacji Tic-Tac-Toe
- Instalacja przykładowych skryptów administracyjnych programu MTS
- Wspomaganie użytkownika w programie MTS

Wymagania systemowe programu MTS

W niniejszym podrozdziale opisano wymagania sprzętowe i programowe instalacji programu MTS, problemy towarzyszące instalacji oraz zagadnienie obsługi programu MTS na platformie systemowej Alpha.

Program MTS można zainstalować na komputerze następującymi metodami:

- Korzystając z pakietu Windows NT 4.0 Option Pack w celu instalacji programu MTS razem z programem Internet Information Server (IIS) oraz innymi składnikami pakietu.
- Korzystając z pakietu Windows NT 4.0 Option Pack w celu instalacji programu MTS bez żadnych innych składników pakietu.

Program MTS może być używany w systemach Windows NT oraz Windows 95 zapewniających obsługę modelu DCOM.

Aby zainstalować program MTS bez innych składników pakietu Option Pack:

1. Uruchom program instalacyjny pakietu Option Pack. Wybierz opcję instalacji **Standardowa**.
2. Anuluj zaznaczenia wszystkich składników pakietu Option Pack.
3. Zaznacz pole wyboru dla programu Transaction Server (nie klikaj w polu).
4. Kliknij przycisk **Pokaż elementy składowe**.
5. Zaznacz pole **Składniki rdzenia programu Transaction Server**. W ten sposób zostanie również wybrany program Microsoft Management Console.

   Zaznaczenie pola **Development Option** powoduje instalację składników dostępu do danych.
6. Kliknij przycisk **OK**, aby kontynuować instalację.

Wymagania sprzętowe

Przed rozpoczęciem instalacji programu Microsoft Transaction Server należy upewnić się, czy komputer spełnia następujące minimalne wymagania sprzętowe:

- Komputer zgodny ze standardem i386, wyposażony w system Windows NT lub Windows 95 z obsługą modelu DCOM lub komputer typu Alpha AXP™. Najnowsze informacje o wymaganiach sprzętowych można uzyskać w dokumencie "MTS Release Notes".
- Dysk twardy zapewniający co najmniej 30 MB dostępnej pamięci (dla potrzeb pełnej instalacji).
- Stacja dysków CD-ROM.
- Dowolny monitor obsługiwany przez system Windows NT w wersji 4.0 lub Windows 95.
- Co najmniej 32 MB pamięci.
- Mysz lub inne dopuszczalne urządzenie wskazujące.

Za pomocą programu instalacyjnego pakietu Windows NT 4.0 Option Pack program MTS można zainstalować bezobsługowo. Przed rozpoczęciem instalacji w trybie bezobsługowym należy zmodyfikować plik instalacyjny zgodnie z wybranymi opcjami instalacji. Plik instalacji bezobsługowej (unattend.txt) znajduje się w katalogu \MTS.

Aby przeprowadzić instalację w trybie bezobsługowym, należy kliknąć przycisk **Start**, wybrać polecenie **Uruchom** i wpisać, co następuje:

setup /u:unattend.txt.

Wymagania dotyczące oprogramowania

Wymagania dotyczące oprogramowania współpracującego z programem MTS:

- Przed rozpoczęciem instalacji programu MTS na komputerze należy zainstalować system Microsoft Windows NT w wersji 4.0 lub nowszej albo system Windows 95 z obsługą modelu

DCOM. Jeśli na komputerze wyposażonym w system Windows 95 nie zostanie wcześniej zainstalowana obsługa modelu DCOM, w trakcie instalacji programu MTS zostanie wyświetlony następujący komunikat o błędzie:

Nie można pobrać biblioteki Instalatora mtssetup.dll lub nie można odnaleźć funkcji MTSSetupProc.

Obsługę modelu DCOM dla systemu Windows 95 można zainstalować, korzystając z adresu http://www.microsoft.com/oledev. Należy zauważyć, że obsługa modelu DCOM dla systemu Windows 95 jest instalowana przez program Internet Explorer 4.0.

Dezinstalując program MTS, działający w środowisku systemu Windows 95, nie należy odinstalowywać obsługi modelu DCOM. Program Microsoft Distributed Transaction Coordinator (MS DTC), który jest instalowany, ale nie usuwany przez instalator programu MTS, wymaga obsługi modelu DCOM.

- Jeśli występuje konieczność zdalnego administrowania komputera wyposażonego w system Windows NT za pomocą komputera wyposażonego w system Windows 95, należy zainstalować usługę Rejestr zdalny dla systemu Windows 95. Usługa Rejestr zdalny pozwala zmieniać wpisy do rejestru zdalnych komputerów wyposażonych w system Windows NT (pod warunkiem posiadania odpowiednich uprawnień). Pliki usługi Rejestr zdalny znajdują się na dysku CD Windows 95, w podkatalogu \Admin\Nettols\RemoteReg. Informacje na temat procedury jej instalowania zawiera plik Regserv.txt.
- Aby w ramach programu MTS móc współpracować z klientami wyposażonymi w system Microsoft Windows 95, należy zainstalować obsługę modelu DCOM dla systemu Windows 95. Najnowsze informacje na temat obsługi modelu DCOM dla systemu Windows 95 można znaleźć pod adresem http://www.microsoft.com/oledev/.
- W przypadku korzystania z systemu Windows NT Server, należy zainstalować pakiet Windows NT Service Pack 3. Pakiet Windows NT Service Pack 3 można pobrać spod adresu http://www.microsoft.com/support.
- Aby wykorzystywane składniki miały dostęp do baz danych, należy używać programu Microsoft SQL Server w wersji 6.5 (lub nowszej) albo innego programu do obsługi baz danych zgodnego z programem Microsoft Transaction Server. Tu trzeba zauważyć, że program SQL Server wymaga systemu operacyjnego Windows NT.
- Zaleca się, aby na komputerach korzystających z programu MTS był zainstalowany pakiet SQL Server Service Pack 3. Dzięki temu można uniknąć wielu znanych problemów. Pakiet SQL Server Service Pack 3 można zainstalować spod adresu http://www.microsoft.com/support/.
- Użytkownicy zamierzający tworzyć składniki za pomocą programu Microsoft Visual Basic powinni korzystać z programu Microsoft Visual Basic Enterprise Edition, w wersji 4.0 lub nowszej. Do budowania składników programu MTS zaleca się stosowanie wersji 5.0 programu Visual Basic.
- Użytkownicy zamierzający tworzyć składniki za pomocą programu Microsoft Visual C++® powinni korzystać z wersji 4.1 lub nowszej tego programu oraz programu Active Template Libraries (ATL) w wersji 1.1 lub nowszej. W przypadku wersji 4.1 programu Visual C++, należy zainstalować narzędzie Win32 SDK.
- Użytkownicy zamierzający tworzyć i uruchamiać składniki typu Java przy użyciu programu MTS powinni korzystać z programu Microsoft Virtual Machine for Java, zainstalowanego razem z programem IE 4.0 lub nowszym. Program Internet Explorer można pobrać spod adresu http://www.microsoft.com/ie/.
- Użytkownicy zamierzający tworzyć aplikacje internetowe muszą korzystać z programów Microsoft Internet Information Server w wersji 4.0 lub nowszej oraz z programu Microsoft Internet Explorer w wersji 4.0 lub nowszej.
- W celu zapewnienia współpracy programu MTS z bazami danych programów Oracle 7.3.3 lub Oracle 8 należy zainstalować Oracle patch 7.3.3.2.0 **Error! Reference source not found.** Oracle. Aby uzyskać this patch należy skontaktować się z przedstawicielstwem firmy Oracle. Co więcej, warto zapoznać się z tematem Ustawianie programu MTS dla potrzeb współpracy z programem Oracle, który zawiera wyczerpujące informacje na temat zapewniania współpracy programów Oracle i Microsoft Transaction Server.

Korzystanie z programu Advanced Data Connector 1.1

W ramach pakietu Windows NT 4.0 Option Pack jest instalowany program Advanced Data Connector (ADC) w wersji 1.5. Klienci programu ADC 1.5 wymagają zainstalowania programu Internet Explorer (IE) 4.0. Klienci programu ADC 1.5 nie są w stanie współpracować z żadną wcześniejszą wersją programu Internet Explorer niż 4.0.

Użytkownicy przykładowej aplikacji MTS Adventure Works powinni skorzystać z adresu http://www.microsoft.com/adc, aby uzyskać w ten sposób uaktualnioną wersję programu ADC 1.5, który może współpracować z programem Internet Explorer w wersji 3.0 (lub nowszej). Aby utworzyć odwołanie do nowej biblioteki rekordów ADC 1.5 (Msador15.dll), należy ponownie skompilować aplikację MTS Adventure Works.

Instalacja programu SQL Server z programem MTS

Jeśli na komputerze mają być instalowane programy SQL Server i MTS, program SQL Server należy zainstalować jako pierwszy. W razie późniejszej ponownej instalacji programu SQL Server trzeba ponownie zainstalować program MTS. Niedogodność ta zostanie wyeliminowana w przyszłym wydaniu programu SQL Server.

Instalacja programu MTS 1.0 w miejsce programu MTS 2.0

Aby na komputerze, na którym zainstalowano już program MTS 2.0, zainstalować program MTS 1.0, należy usunąć z katalogu systemowego następujące pliki (przed uruchomieniem instalatora programu MTS 1.0):

%WINDIR%\system32\adme.dll
%WINDIR%\system32\dac.exe
%WINDIR%\system32\dacdll.dll
%WINDIR%\system32\dtccm.dll
%WINDIR%\system32\dtctrace.dll
%WINDIR%\system32\dtctrace.exe
%WINDIR%\system32\dtcuic.dll
%WINDIR%\system32\dtcuis.dll
%WINDIR%\system32\dtcutil.dll
%WINDIR%\system32\dtcxatm.dll
%WINDIR%\system32\enudtc.dll
%WINDIR%\system32\logmgr.dll
%WINDIR%\system32\msdtc.exe
%WINDIR%\system32\msdtc.dll
%WINDIR%\system32\msdtcprx.dll
%WINDIR%\system32\msdtctm.dll
%WINDIR%\system32\dtccfg.cpl
%WINDIR%\system32\svcsrvl.dll
%WINDIR%\system32\xolehlp.dll
%WINDIR%\system32\mmc.exe
%WINDIR%\system32\mmc.ini
%WINDIR%\system32\mmclv.dll
%WINDIR%\system32\mmcndmgr.dll
%WINDIR%\system32\mtxinfr1.dll
%WINDIR%\system32\mtxinfr2.dll
%WINDIR%\system32\mtxclu.dll

%WINDIR%\system32\mtxrn.dll

%WINDIR%\system32\mtxdm.dll

Opcja dezinstalacji powoduje usunięcie pakietów zdefiniowanych przez użytkownika

Jeśli na komputerze jest zainstalowana poprzednia wersja programu MTS, a zachodzi konieczność zachowania pakietów zdefiniowanych przez użytkownika, wówczas nie należy przeprowadzać procedury dezinstalacji; ani za pomocą opcji **Odinstaluj**, ani za pomocą ikony **Dodaj/Usuń programy** w Panelu sterowania. Odinstalowanie programu MTS spowoduje usunięcie wszystkich pakietów zdefiniowanych przez użytkownika. Aby podczas uaktualniania programu zachować te pakiety, należy zainstalować program MTS w miejsce jego istniejącej wersji.

Nieprawidłowe hasła chroniące tożsamość pakietu systemowego

Jeśli w trakcie instalacji programu MTS zostanie nieprawidłowo podane hasło dla tożsamości pakietu systemowego, program Microsoft Transaction Server Explorer nie będzie działał. Serwer katalogu (pakiet systemowy) zwróci błąd o numerze 80008005 (awaria pliku wykonywalnego serwera). W celu poprawienia nieprawidłowego hasła systemowego należy ponownie zainstalować program Microsoft Transaction Server.

Oprócz tego, podanie niepoprawnego hasła w polu **Tożsamość pakietu systemowego** spowoduje niemożność uruchomienia pakietu. Zostanie zwrócony ten sam błąd nr. 80008005. Tym razem jednak, klient będzie informowany o awarii za pomocą odpowiedniej wartości parametru HRESULT obiektu **CoCreateInstance** (lub równoważnego). Aby poprawić ten błąd, należy powrócić do programu MTS Explorer i zmodyfikować tożsamość pakietu (za pomocą karty **Tożsamość** pakietu). W ten sposób zostaną zweryfikowane identyfikatory użytkowników, ale nie zostaną zweryfikowane hasła (z powodu ograniczeń zabezpieczeń w systemie Windows NT).

Platformy sprzętowe Alpha

Program MTS może być instalowany na komputerach typu Alpha AXP™. Niemniej jednak, niniejsze wydanie programu Microsoft Transaction Server dla komputerów Alpha nie obejmuje następujących elementów i funkcji:

- Składniki i przykłady programów utworzone w programie Microsoft Visual J++
- Obsługa dostępu do baz danych Oracle

Zarówno przykładowe składniki utworzone za pomocą programów Microsoft Visual C++ i Visual Basic, jak i przykładowe skrypty administracyjne znajdują się w folderze \Samples.

Najnowsze informacje na temat obsługi platformy sprzętowej Alpha można uzyskać pod adresem http://www.microsoft.com/transaction.

Aby sprawdzić poprawność instalacji programu MTS na platformie sprzętowej Alpha, należy uruchomić przykładową aplikację Sample Bank, wykorzystując do tego dwa komputery: komputer typu Alpha (jako serwer) i komputer typu X86 (jako klient). Bardziej szczegółowe informacje na ten temat można uzyskać pod hasłem Instalacja przykładowej aplikacji Sample Bank.

Instalacja dokumentacji i przykładów projektowania w programie MTS

Dokumentację i przykłady projektowania w programie Microsoft Transaction Server (MTS) można zainstalować, wybierając opcję instalacji **Niestandardowa**. Poniżej zestawiono składniki programu MTS instalowane dla poszczególnych opcji instalacyjnych.

- **Minimalna**

  Opcja powoduje zainstalowanie środowiska czasu wykonywania programu MTS oraz programu MTS Explorer.

- **Typowa**

  Opcja powoduje zainstalowanie środowiska czasu wykonywania programu MTS, programu MTS Explorer oraz podstawowej dokumentacji programu MTS.

- **Niestandardowa**

  Opcja powoduje zainstalowanie środowiska czasu wykonywania programu MTS, programu MTS Explorer, podstawowej dokumentacji programu MTS, przykładów projektowania oraz dokumentacji związanej z projektowaniem w programie MTS.

Użytkownicy mniej doświadczeni w posługiwaniu się programem MTS powinni zawsze wybierać opcję instalacji **Niestandardowa** i instalować w ten sposób dokumentację oraz przykłady związane z projektowaniem. Zapoznanie się z nimi pozwoli lepiej zrozumieć zagadnienia zarządzania pakietami i składnikami. Na przykład aplikacje przykładowe Sample Bank i Tic-Tac-Toe stwarzają możliwość przećwiczenia procedur rozmieszczania pakietów i administrowania pakietami za pomocą programu MTS Explorer. Ponadto aplikacje przykładowe pozwalają sprawdzić poprawność instalacji programu MTS.

W dokumentacji projektowania programu MTS opisano czynności projektowe określające sposób łączenia składników w pakiety, na przykład ustawianie właściwości aktywacji i właściwości transakcyjnych. Oprócz omówienia pojęć typowo programistycznych (na przykład stanu współużytkowania) podręcznik *MTS Programmer's Guide* zawiera również opis instalacji i konfigurowania aplikacji Sample Bank.

Aby zainstalować przykłady projektowania i dokumentację dotyczącą projektowania po uprzedniej minimalnej lub typowej instalacji programu MTS:

1. Otwórz menu **Start**, kliknij polecenie **Ustawienia**, a następnie **Panel sterowania**.
2. Zaznacz ikonę **Dodaj/Usuń programy**, kliknij pozycję **Microsoft Transaction Server**, a następnie przycisk **Dodaj/Usuń**.

   Należy pamiętać, że odinstalowanie programu MTS spowoduje usunięcie pakietów zdefiniowanych przez użytkownika. Aby zachować te pakiety, zainstaluj program MTS bez usuwania poprzedniej instalacji.
3. W programie instalacyjnym kliknij przycisk **Zainstaluj ponownie/Dodaj**.
4. Zaznacz opcję **Microsoft Transaction Server**, a następnie kliknij przycisk **Pokaż elementy składowe**.
5. Sprawdź, czy zaznaczone są wszystkie pola wyboru elementów składowych.
6. Kliknij przycisk **OK**.

Konfigurowanie serwera programu MTS

Po zainstalowaniu programu MTS należy skonfigurować serwer MTS tak, aby stało się możliwe rozmieszczanie i zarządzanie pakietami za pomocą programu MTS Explorer. Przed rozpoczęciem rozmieszczania i instalowania należy wykonać następujące czynności konfiguracyjne:

- Skonfigurować role i tożsamość dla pakietu systemowego
- Ustawić komputery do administrowania

Konfigurowanie ról dla pakietu systemowego

Bezpieczne rozmieszczanie pakietów MTS i zarządzanie nimi wymaga mapowania roli "Administrator" pakietu systemowego do odpowiedniego użytkownika. Po zainstalowaniu programu MTS do roli administratora pakietu systemowego nie są przypisani żadni użytkownicy. Z tego względu zabezpieczenia pakietu systemowego są wyłączone i żaden użytkownik nie ma możliwości modyfikowania (za pomocą programu MTS Explorer) na tym komputerze konfiguracji pakietu systemowego. Dopiero po przypisaniu użytkowników do ról pakietu systemowego program MTS będzie obsługiwał zabezpieczenia, to znaczy przy każdej próbie modyfikacji pakietów za pomocą programu MTS Explorer będzie sprawdzał role.

Domyślnie, pakiet systemowy ma role "Administrator" i "Odczyt". Użytkownicy przypisani do roli "Administrator" pakietu systemowego mogą korzystać z dowolnych funkcji programu MTS Explorer. Użytkownicy przypisani do roli "Odczyt" mogą przeglądać obiekty w hierarchii obiektów programu MTS, ale nie mogą instalować, tworzyć, zmieniać i usuwać obiektów, nie mają także uprawnień do zamykania procesów serwera i eksportowania pakietów. Na przykład jeśli dany użytkownik zamapuje swoją nazwę użytkownika domeny systemu Windows NT do roli "Administrator" pakietu systemowego, zyska możliwość dodawania, modyfikowania i usuwania dowolnych pakietów (za pomocą programu MTS Explorer). Jeśli program MTS jest zainstalowany na serwerze, którego rolą jest podstawowy lub zapasowy kontroler domeny, wówczas uprawnienia do zarządzania pakietami (za pomocą programu MTS Explorer) mają tylko administratorzy domeny.

Więcej informacji na temat mapowania użytkowników do ról można uzyskać pod hasłem Mapowanie ról MTS do użytkowników i grup.

Dla pakietu systemowego można tworzyć także nowe role. Na przykład można utworzyć rolę "Projektant", pozwalającą instalować i uruchamiać pakiety, ale nie pozwalającą usuwać ich i eksportować. Zamapowane do tej roli grupy lub użytkownicy systemu Windows NT będą w stanie testować instalację pakietów na komputerze, nawet bez uprawnień administratora tego komputera. Więcej informacji na temat tworzenia nowych ról można uzyskać pod hasłem Dodawanie nowej roli MTS.

Po skonfigurowaniu wszystkich ról dla pakietu systemowego, należy włączyć sprawdzanie upoważnień, to znaczy przejść do arkusza właściwości zabezpieczeń pakietu i zaznaczyć odpowiednie pole wyboru. Wyczerpujące informacje na temat włączania sprawdzania upoważnień można znaleźć pod hasłem Włączanie zabezpieczeń pakietu MTS.

Należy zauważyć, że poniższe funkcje administracyjne usługi MS DTC nie wymagają użycia pakietu systemowego i roli pakietu systemowego:

- okno **Statystyki transakcji**
- okno **Lista transakcji**
- okno **Komunikaty śledzenia**
- polecenia **Start/Zatrzymaj** usługi MS DTC

Ustawianie komputerów dla potrzeb administrowania za pomocą programu MTS Explorer

Domyślnie komputer, na którym zainstalowano program MTS, jest zarządzany za pomocą programu MTS Explorer pod nazwą “Mój komputer”. Program MTS Explorer pozwala zarządzać także innymi komputerami. Wszystkie nowe komputery należy dodać do folderu "Komputery", to znaczy zaznaczyć ikonę komputera i wykonać jedną z poniższych czynności:

- Wybrać polecenie **Nowy** z menu **Akcja**
- Kliknąć ikonę **Utwórz nowy obiekt** z paska narzędzi programu MTS Explorer
- Kliknąć prawym przyciskiem myszy komputer o nazwie "Mój komputer", a następnie wybrać polecenia **Nowy** i **Komputer**

Następnie, w oknie dialogowym, należy wprowadzić nazwę komputera z używanej domeny systemu Windows NT; komputer ten zostanie dodany jako komputer zdalny i będzie reprezentowany przez folder najwyższego poziomu. Użytkownik powinien być zamapowany do roli administratora komputera zdalnego.

Więcej informacji na temat zarządzania obiektami w hierarchii programu MTS Explorer można uzyskać pod hasłem Hierarchia programu MTS Explorer.

Ważne Zdalne administrowanie komputerami wyposażonymi w system Windows 95 za pomocą programu MTS działającego na serwerach wyposażonych w system Windows NT jest niemożliwe.

Konfigurowanie programu MTS z programem Microsoft Cluster Server

Microsoft Cluster Server (MSCS) jest programem do obsługi klastrów złożonych z komputerów korzystających z systemu Windows NT Server. Program MSCS w wersji 1.0 obsługuje klastry dwóch specjalnie połączonych serwerów korzystających z systemu Windows NT Server. Jeśli jeden z serwerów klastra nie będzie w stanie wykonać pewnej operacji lub przejdzie do trybu "offline", jego funkcje przejmie zastępczo drugi serwer.

Aby program Microsoft Transaction Server mógł współpracować z programem MSCS, należy wcześniej zainstalować na komputerze system Windows NT Server 4.0 Enterprise. Następnie trzeba zainstalować program Microsoft Cluster Server 1.0. Ostatecznie, należy zainstalować program Microsoft Transaction Server 2.0 (zgodnie z przedstawionym dalej opisem).

Uwaga Obsługa klastrów w programie MTS 2.0 nie jest możliwa w przypadku zainstalowania wstępnych wersji (ang. pre-release versions) programu Microsoft Cluster Server 1.0. Instalacja programu MTS 2.0 ze wstępną wersją programu Microsoft Cluster Server nie zapewnia prawidłowego działania programu MSCS.

Więcej informacji na temat programu MSCS można uzyskać w dokumentacji dostępnej w trybie online systemu Windows NT (ang. Windows NT Books Online).

▶ Aby zainstalować program Microsoft Transaction Server 2.0 z programem Microsoft Cluster Server 1.0

1. Zainstaluj system Windows NT Server 4.0 Enterprise.
2. Zainstaluj program MSCS 1.0 na wszystkich komputerach klastra.
3. Za pomocą narzędzia MSCS Cluster Administrator skonfiguruj grupę tak, aby zawierała zasób Nazwa sieci i zasób Dysk współużytkowany.
4. Zainstaluj Microsoft Transaction Server 2.0 w węźle, do którego przypisana jest grupa skonfigurowana powyżej. Więcej informacji na ten temat można uzyskać pod hasłem Instalowanie programu Microsoft Transaction Server.
5. Kiedy program MTS wykryje w systemie program MSCS, zostanie wyświetlone okno dialogowe z pytaniem o nazwę serwera wirtualnego, na którym będzie instalowany program Microsoft Distributed Transaction Coordinator. Podaj nazwę zasobu Nazwa sieci, skonfigurowanego w kroku 3.
6. W tym samym oknie dialogowym podaj lokalizację piku dziennika usługi MS DTC na dysku współużytkowanym, skonfigurowanym w kroku 3.
7. Kliknij przycisk **DALEJ**, aby kontynuować instalację programu MTS.
8. Zainstaluj program MTS na drugim komputerze klastra. Podczas instalacji nie trzeba będzie podawać lokalizacji serwera wirtualnego i pliku dziennika.

Instalacje programu MTS nie powinny być uruchamiane równolegle, w różnych węzłach klastra. Należy zainstalować program w pierwszym węźle, a następnie w drugim bez ponownego rozruchu pierwszego węzła. Dopiero po zainstalowaniu programu MTS w obydwu węzłach można wykonać ich ponowny rozruch.

Uwaga Przed instalacją programu Microsoft Transaction Server 2.0 z programem Microsoft Cluster Server 1.0 nie należy odinstalowywać programu Microsoft Transaction Server 1.0. W celu zainstalowania programu MTS 2.0 z programem MCS 1.0 należy wykonać poniższe czynności.

▶ Aby uaktualnić program Microsoft Transaction Server 1.1 do programu Microsoft Transaction Server 2.0 z narzędziem Microsoft Cluster Server

1. Usuń wszystkie zasoby pakietów programu Microsoft Transaction Server, które utworzono za pomocą programu Microsoft Transaction Server 1.1 przy współpracy z programem Microsoft Cluster Server.
2. Uruchom instalatora programu Microsoft Transaction Server 2.0 w węźle klastra, do którego należy

zasób Microsoft Distributed Transaction Coordinator. W ten sposób aktualizacja węzła zostanie wykonana automatycznie. Wykonanie następnego kroku wymaga zakończenia instalacji pierwszego programu Microsoft Transaction Server.

3 Uruchom program instalacyjny programu Microsoft Transaction Server 2.0 w drugim węźle klastra.

4 Wykonaj ponowny rozruch obydwu klastrów.

Dezinstalacja programu Microsoft Cluster Server na serwerze programu MTS

Aby odinstalować program Microsoft Cluster Server w jednym z węzłów klastra, wykonaj następujące kroki:

1 Za pomocą Administratora klastrów przełącz usługę Microsoft Distributed Transaction Coordinator do trybu "offline".

2 Odinstaluj program Microsoft Cluster Server z węzła i wykonaj ponowny rozruch węzła.

3 Za pomocą programu Transaction Server Explorer zmień lokalizację pliku dziennika usługi Microsoft Distributed Transaction Coordinator. Plik dziennika musi znajdować się na dysku niewspółużytkowanym. Nie spełnienie tego warunku może spowodować uszkodzenie pliku dziennika lub naruszenia dostępu w usłudze Microsoft Distributed Transaction Coordinator.

Resetowanie pliku dziennika usługi MS DTC na serwerach sklastrowanych

Aby zrestować plik dziennika usługi MS DTC na serwerze sklastrowanym, należy uruchomić program MTS Explorer w węźle, który aktualnie obejmuje dysk współużytkowany zawierający plik dziennika.

Uruchamianie i zatrzymywanie usługi MS DTC na serwerach sklastrowanych

Uruchamianie i zatrzymywanie usługi MS DTC na serwerach sklastrowanych umożliwia program MTS Explorer oraz narzędzie administracyjne usługi MSCS. Nie można natomiast używać poleceń "net start msdtc" i "net stop msdtc". W celu uruchmienia lub zatrzymania usługi MS DTC (w klastrze MSCS) z poziomu wiersza poleceń należy wpisać w nim odpowiednio: "msdtc -start" lub "msdtc -stop".

Wywoływanie obiektów programu MTS po awarii programu MSCS

Aplikacje klientów programu Microsoft Transaction Server muszą bezwzględnie zwolnić wszystkie odwołania do uszkodzonegho węzła i utworzyć ponownie nowe instancje składników. Instancje składników są tworzone w drugim węźle.

Korzystanie z folderu "Zdalne składniki" na serwerach sklastrowanych

Za pomocą folderu "Zdalne składniki" można ściągnąć dane składniki z węzła usługi Microsoft Cluster Server (MSCS). W tym celu wykonaj następujące kroki:

1 Na serwerze, z którego będą ściągane składniki kliknij prawym przyciskiem myszy ikonę **Mój komputer** i wybierz polecenie **Właściwości**.

2 Kliknij kartę **Opcje**, a następnie w polu **Nazwa zdalnego serwera** wprowadź nazwę serwera wirtualnego. Kliknij przycisk **OK**.

3 Na komputerze klienta, do którego będą przekazywane składniki dodaj komputer z zainstalowanym programem MSCS do folderu "Komputery", używając fizycznej nazwy serwera.

4 Zaznacz folder "Zdalne składniki" i wybierz polecenie **Nowy** z menu **Akcja**. Możesz również kliknąć folder "Zdalne składniki" prawym przyciskiem myszy, a następnie wybrać z menu podręcznego polecenia **Nowy** i **Składnik**.

5 Wykonuj instrukcje kreatora instalacji składników zdalnych. W trakcie instalacji składników zdalnych jest wyświetlana nazwa fizyczna serwera, a nie nazwa wirtualna (zgodnie z zawartością pola **Nazwa zdalnego serwera**, ustawianego po stronie serwera).

6 Zaznacz folder "Zdalne składniki" i kliknij przycisk **Właściwości**, wyświetlany na pasku narzędzi programu MTS Explorer. W oknie dialogowym **Właściwości**, w kolumnie **Serwer** będzie

wyświetlana wirtualna nazwa serwera.

Uwaga Jeśli na komputerze, na którym działa program MSCS, zostanie zmieniona nazwa serwera zdalnego, składniki zdalne należy zainstalować ponownie również na komputerach klientów.

Replikacja pustych pakietów

W programie MTS nie można replikować pustych pakietów.

Ustawianie programu MTS dla potrzeb współpracy z programem Oracle

Transakcyjne składniki programu MTS można zaprojektować w sposób zapewniający im dostęp do baz danych programu Oracle 7.3.3 za pośrednictwem interfejsów ODBC. Program MTS może współpracować z programami: Oracle 7 Workgroup Server dla systemu Windows NT, Oracle 7 Enterprise Server dla systemu Windows NT, Oracle 7 Enterprise Servers dla systemu UNIX oraz Oracle Parallel Server dla systemu UNIX.

Tworzone składniki MTS mogą uzyskiwać dostęp do baz danych programu Oracle 8, działających zarówno w systemie Windows NT, jak i Unix. W tym celu wystarczy, aby składnik korzystał z oprogramowania klienckiego programu Oracle 7. Program MTS nie obsługuje oprogramowania klienckiego programu Oracle 8.

W niniejszym podrozdziale omówiono następujące tematy:

Wymagane oprogramowanie

Instalacja obsługi baz danych Oracle

Testowanie instalacji i konfiguracji dla baz danych Oracle

Sprawdzanie poprawności instalacji i konfiguracji programu Oracle za pomocą przykładowej aplikacji Sample Bank

Znane ograniczenia we współpracy programów MTS i Oracle

Wymagane oprogramowanie

W poniższej tabeli zestawiono oprogramowanie niezbędne do tego, aby składniki programu MTS (działające na platformie systemowej Windows NT lub UNIX) mogły uzyskiwać dostęp do baz danych Oracle.

Składnik	**Wersja**
Oracle for Windows NT	7.3.1 (with patch 2 or later)
Oracle SQL*Net	2.3.3
Oracle OCIW32.DLL	1, 0, 0, 5
Oracle for UNIX	7.3.1 (with patches)
Microsoft Transaction Server 2.0	2.0
Microsoft ODBC Driver for Oracle (MSORCL32.DLL)	2.0
ActiveX Data Objects (ADO)	1.5

Ważne Wcześniejsze wersje oprogramowania nie będą działać poprawnie. Proszę upewnić się, czy zainstalowano prawidłową wersję programu. Niespełnienie tego warunku jest jak dotychczas najczęstszą przyczyną niewłaściwej współpracy między programami MTS i Oracle.

Program Oracle dla systemu Windows NT

Należy zainstalować albo wersję Oracle 7.3.3 Workgroup Server dla systemu Windows NT albo wersję Oracle 7.3.3 Enterprise Server dla systemu Windows NT. Program Oracle 7.3.2 oraz wersje wcześniejsze dla systemu Windows NT nie są obsługiwane, a zatem nie będą współpracować z transakcjami MTS.

Należy zainstalować program Oracle 7.3.3 patch release 2 or later. This patch jest wymagany dla wszystkich klientów programu Oracle 7.3.3 korzystających z baz danych programów Oracle 7.3.3 lub Oracle 8. Oracle patch release 2 zawiera szereg poprawek, które są niezbędne do właściwej obsługi transakcji XA w systemie Windows NT. Program Oracle 7.3.3 nie będzie współpracował z programem

MTS, o ile na komputerze nie zostanie zainstalowany Oracle 7.3.3 patch release 2.

Uwaga Jeśli podczas instalacji Oracle patch release 2 w systemie Windows 95 wystąpia problemy, proszę skontaktować się z firmą Oracle.

Aby uzyskać Oracle 7.3.3 patch releases z Działu Obsługi Klienta firmy Oracle (ang. Oracle Customer Support Organization), należy wysłać do firmy raport o występujących problemach. Do chwili opracowania niniejszej dokumentacji ???these patch releases nie były jeszcze dostępne w sieci Web, w publicznej witrynie firmy Oracle.

Oracle SQL*Net

Należy zainstalować program Oracle SQL*Net 2.3.3 dla systemu Windows NT. Tę wersję programu można uzyskać od firmy Oracle. Wcześniejsze wersje programu Oracle SQL*Net mogą nie działać.

Składnik OCIW32.DLL firmy Oracle

Należy upewnić się, czy zainstalowano poprawną wersję składnika OCIW32.DLL firmy Oracle. Kwestia poprawności wersji jest niezwykle ważna.

Poprawna wersja składnika OCIW32.DLL ma następujące parametry:

Version 1, 0, 0, 5
Tuesday, March 18, 1997 2:47:52 PM
Size 18KB.

Niepoprawna wersja składnika OCIW32.DLL ma następujące parametry:

Version 7.x
Thursday, February 01, 1996 12:50:06 AM
Size 36 KB

Poprawną wersję powyższej biblioteki DLL można uzyskać z instalacyjnego dysku CD programu Oracle 7.3.3, zlokalizowanego w katalogu \WIN32\V7\RSF73.

Bazy danych firmy Oracle dla systemu UNIX

Aby w systemie UNIX transakcyjne składniki programu MTS mogły uzyskiwać dostęp do baz danych Oracle, należy zainstalować program Oracle w wersji 7.3.3 (lub nowszej) dla używanej platformy systemowej UNIX. W większości przypadków niezbędne jest zainstalowanie Oracle 7.3.3 patch release for Oracle on UNIX.

Aby sprawdzić, czy dla danej platformy systemowej UNIX jest niezbędny Oracle 7.3.3 patch release, należy skontaktować się z Działem Obsługi Klienta firmy Oracle (ang. Oracle Customer Support). Należy wyjaśnić, że chodzi o możliwość dostępu do bazy danych Oracle w systemie UNIX za pomocą nowego standardu obsługi transakcji XA, który stanowi element programu Oracle 7.3.3 dla systemu Windows NT.

Zgodnie z naszą aktualną wiedzą następujące patch releases działają prawidłowo:

Platforma	Oracle Patch
HP 9000	7.3.3.3
IBM AIX	7.3.3.2
Sun Solaris	7.3.3.2

Program Microsoft Transaction Server 2.0

Dostęp do baz danych Oracle za pomocą programu MTS wymaga zainstalowania programu Microsoft Transaction Server 2.0.

Sterowniki Microsoft ODBC dla programu Oracle

Współpraca z bazami danych Oracle wymaga zainstalowania sterownika Microsoft ODBC 2.0 dla programu Oracle (MSORCL32.DLL). Instalator pakietu Windows NT 4.0 Option Pack instaluje tę bibliotekę DLL automatycznie.

Aby uzyskiwać dostęp do baz danych Oracle, zaleca się korzystanie z nowego sterownika Microsoft ODBC dla programu Oracle 2.0, nawet jeśli nie jest niezbędna obsługa transakcji. Ten nowy sterownik zapewnia większą wydajność niż jego poprzednik, sterownik ODBC 1.0. Sterownik ODBC 1.0 działa na zasadzie szeregowania operacji na poziomie sterownika; wszystkie żądania są umieszczane w jednym wątku sterownika. Sterownik ODBC 2.0 natomiast szereguje wszystkie operacje na poziomie połączenia. Dzięki temu różne połączenia z bazą danych mogą być używane równolegle.

Obiekty danych ADO (z ang. ActiveX Data Objects)

Jeśli używane aplikacje korzystają z obiektów ADO, należy zainstalować program ADO w wersji 1.5. Wcześniejsze wersje programu ADO nie współpracują z nowym programem ODBC 3.5 Driver Manager. Program ADO 1.5 jest zawarty w programie instalacyjnym pakietu NT 4.0 Option Pack.

Instalacja obsługi baz danych Oracle

Aby zainstalować obsługę baz danych Oracle dla transakcyjnych składników MTS

1 Zainstaluj program Oracle 7.3.3 dla systemu Windows NT.

Jeśli wykorzystywana baza danych Oracle działa w systemie UNIX, zainstaluj program Oracle 7.3.3 dla tego systemu.

2 Zainstaluj Oracle 7.3.3 patch 2 or later dla systemu Windows NT. Wynikową wersją programu Oracle będzie Oracle 7.3.3.2 lub nowsza w zależności od instalowanego Oracle patch. Aby korzystać z baz danych Oracle 7 lub Oracle 8 (w systemie Windows NT lub Unix) należy zainstalować Oracle 7.3.3 patch 2 or later. These Oracle patches uwalniają klientów programu Oracle od dotychczasowych problemów.

Jeśli korzystasz z systemu UNIX, zainstaluj dowolne Oracle 7.3.3 patch releases, wymagane w używanym systemie UNIX.

3 Upewnij się, czy zainstalowana jest poprawna wersja składnika OCIW32.DLL, zgodnie z opisem w podrozdziale Wymagane oprogramowanie.

4 Zainstaluj program Microsoft Transaction Server 2.0 w wersji 3.0, który automatycznie instaluje następujące składniki:

- Program Microsoft Transaction Server 2.0 razem z interfejsem Microsoft OCI
- Sterownik Microsoft ODBC 3.5
- Sterownik Microsoft ODBC 2.0 dla programu Oracle
- Program ADO 1.5

1 Usuń plik DTCXATM.LOG. Zlokalizuj ten plik za pomocą Eksploratora. Przed usunięciem pliku DTCXATM.LOG należy zatrzymać usługę Microsoft Distributed Transaction Coordinator.

5 Włącz obsługę standardu Oracle XA.

Aby umożliwić współpracę baz danych Oracle z transakcjami programu MTS

1 Administrator systemu musi utworzyć widoki, określane jako V$XATRANS$. W tym celu administrator powinien uruchomić dostarczony przez firmę Oracle skrypt o nazwie "xaview.sql". Plik skryptu zazwyczaj znajduje się w katalogu C:\ORANT\RDBMS73\ADMIN.

2 Administrator systemu musi udzielić publicznych uprawnień dostępu typu SELECT do utworzonych widoków.

Grant Select on V$XATRANS$ to public.

3 W programie Oracle Instance Manager kliknij polecenie **Advanced Mode** z menu **View**, a następnie w lewym okienku programu zaznacz opcję **Initialization Parameters**.

4 W prawym okienku programu zaznacz opcję **Advanced Tuning** i zwiększ wartość parametru "distributed_transactions". Dzięki temu więcej konkurencyjnych transakcji MTS będzie mogło

jednocześnie aktualizować bazę danych.

Więcej informacji na temat konfiguracji obsługi dla transakcji XA można znaleźć w dokumentacji programu Oracle Server.

Testowanie instalacji i konfiguracji obsługi dla baz danych Oracle

Po zainstalowaniu i skonfigurowaniu obsługi dla baz danych Oracle, należy sprawdzić poprawność przeprowadzonej instalacji. Służy do tego specjalny program testujący, instalowany razem z programem MTS. Program testujący, podobnie jak sam program MTS, wykorzystuje interfejsy OCI XA firmy Oracle.

Program testujący pozwala sprawdzić, czy z bazami danych Oracle można połączyć się za pomocą specjalnej funkcji do obsługi transakcji XA. Program testujący wykorzystuje standardowe interfejsy firmy Oracle oraz funkcje do obsługi transakcji XA. Nie korzysta natomiast z programów Microsoft Transaction Server i Microsoft Distributed Transaction Coordinator. Z tego względu niepomyślny wynik programu testującego wskazuje na nieprawidłową konfigurację lub instalację programu Oracle. W razie takiej ewentualności należy ponownie zainstalować go i skonfigurować lub skontaktować się z przedstawicielem firmy Oracle.

Aby uruchomić program testujący firmy Oracle

1. Sprawdź, czy zainstalowano poprawne wersje oprogramowania opisanego w podrozdziale Wymagane oprogramowanie.
2. Utwórz źródło danych ODBC DSN odwołujące się do bazy danych Oracle. Upewnij się, czy źródło danych DSN korzysta z nowego sterownika Microsoft Oracle ODBC 2.0.
3. Sprawdź, czy włączono obsługę standardu Oracle XA.
4. Usuń istniejące pliki do śledzenia baz danych Oracle z komputera zawierającego składniki MTS korzystające z baz danych Oracle. W tym celu najprościej jest zlokalizować wszystkie pliki *.TRC za pomocą Eksploratora systemu Windows i usunąć je.
5. Usuń plik DTCXATM.LOG z komputera zarządzającego składnikami MTS, które korzystają z bazy danych Oracle. Plik ten najlepiej odszukać i usunąć za pomocą Eksploratora systemu Windows (jeśli tylko jest on zlokalizowany na Twoim komputerze).
6. W wierszu poleceń trybu MS-DOS uruchom program testujący firmy Oracle (TestOracleXaConfig.exe), a następnie podaj swój identyfikator użytkownika serwera programu Oracle, hasło oraz nazwę serwera. Na przykład:

   ```
   c:>TestOracleXaConfig.exe   -U<user id> -P<Password>
   -S<Server name as in the TNS file>.
   ```

   Jeśli uruchomisz program testujący bez żadnych parametrów, i tak zostaną wyświetlone informacje Pomocy opisujące niezbędne parametry. Program testujący będzie wyświetlał informacje o każdej operacji wykonanej przez program Oracle i oceniał jej poprawność.
7. Jeśli program testujący łączy się z serwerem bazy danych Oracle bez żadnych błędów, oznacza to, że najprawdopodobniej program MTS będzie współpracował z serwerem równie poprawnie. Jeśli program testujący zgłosi jakiekolwiek błędy, wykonaj następujące kroki:
   - Zanotuj dokładnie wyświetlany komunikat o błędzie.
   - Zbadaj plik śledzenia czynności programu Oracle, tworzony w trakcie działań programu testującego. Pliki śledzenia mają rozszerzenie .TRC. Pliki te zawierają wyczerpujące informacje o błędach i są niezwykle użyteczne w diagnozowaniu zaistniałych problemów.
   - Zwróć się o pomoc do przedstawiciela firmy Oracle.

Sprawdzanie poprawności instalacji i konfiguracji programu Oracle za pomocą przykładowej aplikacji Sample Bank

Po sprawdzeniu poprawności instalacji i konfiguracji programu Oracle za pomocą programu testującego można przeprowadzić jeszcze jeden test współpracy programu Microsoft Transaction

Server z bazą danych Oracle, a mianowicie użyć przykładowej aplikacji Sample Bank.

Aby sprawdzić poprawność obsługi bazy danych Oracle za pomocą aplikacji Sample Bank

1. Upewnij się, czy poprawność instalacji i konfiguracji programu Oracle została już sprawdzona przez specjalny program testujący firmy Oracle (instalowany razem z programem MTS).
2. Po stronie serwera utwórz tabelę o nazwie "Account". W poniższym przykładzie pokazano, w jaki sposób określić tabelę.

Owner	**scott**		
Name of Table	**Account**		
Column 1 Name	**AccountNo**	of type	**NUMBER**
Column 2 Name	**Balance**	of type	**NUMBER**

3. Wypełnij tabelę co najmniej dwoma wierszami. Poniżej pokazano przykład wypełnienia tabeli danymi.

AccountNo	Balance
1	1000
2	1000

4. Za pomocą narzędzia do konfigurowania interfejsu ODBC utwórz plik źródła danych DSN. Nadaj plikowi DSN nazwę "MTSSamples". Następnie ręcznie uaktualnij plik DSN, dodając do niego hasło. Poniżej pokazano przykład dodania hasła użytkownika do pliku DSN.

   [ODBC]
   DRIVER=Microsoft ODBC for Oracle
   UID=scott
   PWD=*mypassword*
   ConnectString=*myserver*
   SERVER=*myserver*

5. Zapisz plik DSN i uruchom aplikację kliencką Sample Bank.

Znane ograniczenia we współpracy programów MTS i Oracle

Jeśli program ADO jest używany z programem ODBC 3.5, niezbędna jest wersja Beta programu ADO 1.5

Jeśli używane aplikacje korzystają z programu ADO, należy się upewnić, czy zainstalowano wersję ADO 1.5. Więcej informacji na ten temat można znaleźć w podrozdziale Wymagane oprogramowanie .

Brak obsługi baz danych Oracle na platformie sprzętowej Digital Alpha

Program Microsoft Transaction Server zainstalowany na platformie sprzętowej Digital Alpha nie umożliwia obsługi połączeń z bazami danych Oracle.

Problem wersji biblioteki OCIW32.DLL dla programu Oracle

Niezwykle ważnym warunkiem prawidłowej obsługi baz danych Oracle jest instalacja na komputerze poprawnej wersji biblioteki OCIW32.DLL. Przy każdej nowej instalacji programu MTS lub Oracle należy sprawdzać poprawność wersji tej biblioteki.

Zmiany nazw bibliotek DLL w nowych wersjach programu Oracle

Czasami, po opracowaniu nowej wersji produktu, firma Oracle zmienia nazwy bibliotek DLL. Tymczasem poprawne działanie programu Microsoft Transaction Server zależy od znajomości niektórych nazw tych bibliotek. Obecnie, program MTS poszukuje nazw bibliotek DLL występujących w wersji 7.3.3. programu Oracle. Ponieważ program MTS nie jest w stanie przewidzieć możliwych zmian nazw bibliotek DLL, podczas uaktualniania programu Oracle może wystąpić konieczność zmiany wartości w następującym kluczu rejestru:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Transaction Server\Local Computer\My Computer

W ramach tego klucza występują dwie wartości (w postaci ciągów znakowych).

- OracleXaLib "xa73.dll"
- OracleSqlLib "sqllib18.dll"

Konfiguracja programu Oracle dla potrzeb obsługi dużej liczby połączeń

Aby utworzyć więcej niż kilka połączeń z bazami danych Oracle, należy skonfigurować serwer programu Oracle w sposób zapewniający obsługę dodatkowych połączeń.

Niemożność takiej konfiguracji serwera może wynikać z następujących przyczyn:

- Błędy podczas wywołań **SQLConnect**.
- Problemy z pozyskaniem transakcji wywoływanych obiektów, które mogą powodować następujące błędy w pliku śledzenia programu Oracle:
  - Zbyt wiele sesji
  - Nieudana próba zlokalizowania nazwy serwera przez serwer TNS
  - Zbyt wiele rozpowszechnianych transakcji
- Przekroczenia limitów czasu podczas oczekiwania na blokadę bazy danych. Problemy tego typu występują najczęściej, kiedy skonfigurowana liczba blokad jest niewystarczająca w stosunku do liczby jednocześnie aktywnych transakcji.
- Kolizja rekordów, spowodowana przez blokady występujące dla transakcji wątpliwych.

W przypadku wystąpienia opisanych wyżej problemów, należy zwiększyć wartości następujących parametrów konfiguracyjnych serwera programu Oracle:

- Liczba sesji (przeważnie jest to potrojona liczba oczekiwanych połączeń)
- Liczba rozpowszechnianych transakcji
- Liczba procesów
- Liczba blokad

Instalacja przykładowej aplikacji Sample Bank

Sample Bank jest przykładową aplikacją do obsługi operacji bankowych takich jak kredyty, debety i przekazy pieniężne między różnymi kontami. Uruchamiając aplikację Sample Bank można z jednej strony przetestować poprawność instalacji programu MTS z programem SQL Server 6.5, z drugiej zaś przećwiczyć operacje rozmieszczania pakietów i administrowania pakietami. Aplikacja Sample Bank zawiera składniki napisane w językach programowania Visual Basic, Visual C++ i Visual J++. Jest ona zlokalizowana w podkatalogu \MTS\Samples. Ponadto, program MTS zawiera środowisko Active Server Page (ASP), które w trakcie pracy wywołuje składniki aplikacji Sample Bank. Plik aplikacji nazywa się Bank.asp i znajduje się w podkatalogu \MTS\Samples\asp.

W podręczniku *MTS Programmer's Guide* można znaleźć wyczerpujące informacje na temat budowania składników aplikacji Sample Bank.

W celu uruchomienia aplikacji Sample Bank należy:

- Wybrać opcję instalacji **Niestandardowa** i zaznaczyć wszystkie elementy składowe programu MTS. Więcej informacji na ten temat można uzyskać pod hasłem Instalacja dokumentacji i przykładów projektowania dla programu MTS.
- Zainstalować źródło danych DSN.
- Zainstalować pakiet **Sample Bank**.
- Ustawić program MTS Explorer tak, aby monitorował pakiet **Sample Bank**.
- Uruchomić klienta aplikacji Sample Bank.

Instalacja przykładowej aplikacji Sample Bank

W trakcie instalacji program MTS automatycznie konfiguruje źródło danych ODBC dla potrzeb aplikacji Sample Bank. Ponieważ domyślnie jest wykorzystywany komputer lokalny, na komputerze tym musi być zainstalowany program SQL Server 6.5.

Domyślnie źródło danych (DSN) pogramu MTS wskazuje na program SQL Server 6.5. Aby korzystać z innej bazy danych niż baza programu SQL Server 6.5, należy usunąć domyślne źródło danych DSN i dodać nowe o nazwie MTSSamples, wskazujące na wybraną bazę danych.

Aby korzystać z programu SQL Server zainstalowanego na innym komputerze, należy przejść do Panelu sterowania, użyć ikony usługi ODBC i zmodyfikować źródło danych w sposób następujący:

- W oknie dialogowym **Źródła danych** należy kliknąć kartę **Plik DSN** i zaznaczyć źródło danych MTSSamples.
- Kliknąć przycisk **Konfiguruj** i wprowadzić nazwę wybranego serwera.

Warto zauważyć, że aplikacja Sample Bank nie korzysta z identyfikatora logowania i hasła, określonych przez źródło danych (DSN) MTSSamples. Aplikacja Sample Bank wykorzystuje konto "sa" i hasło puste. Jeśli hasło administratora systemu nie jest pustym ciągiem znaków lub zachodzi konieczność określenia innego identyfikatora logowania, należy zmodyfikować ciąg znaków dla połączenia ODBC, występujący w kodzie źródłowym aplikacji Sample Bank.

Aby monitorować składniki i transakcje pakietu Sample Bank

1. W prawym okienku programu MTS Explorer kliknij dwukrotnie ikonę pakietu **Sample Bank**.
2. Kliknij dwukrotnie folder "Składniki".
3. W menu **Widok** kliknij polecenie **Stan**, aby wyświetlić informacje o wykorzystaniu różnych składników pakietu.
4. W menu **Okno** kliknij polecenie **Nowe okno**.
5. Zmień położenie nowego okna tak, aby obydwa okna nie zachodziły na siebie. Do rozmieszczania okien na ekranie możesz użyć poleceń **Kaskada** lub **Sąsiadująco w poziomie** z menu **Okno**.
6. W lewym okienku nowego okna kliknij przycisk **Statystyki transakcji**.

7 W menu **Akcja** kliknij polecenie **Zakres**, aby wykasować znacznik oraz ukryć lewe okienko. Od tej pory podczas korzystania ze składników transakcyjnych będą wyświetlane statystyki transakcji.

Aby uruchomić klienta Bank Client

1 Upewnij się, czy jest uruchomiona usługa Microsoft Distributed Transaction Coordinator (MS DTC). W lewym okienku programu Transaction Server Explorer zaznacz pozycję **Mój komputer**. Otwórz menu **Akcja** i kliknij polecenie **Uruchom MS DTC** (jeśli jest ono aktywne).

2 Upewnij się, czy jest uruchomiony program SQL Server. Program ten można uruchomić z Panelu sterowania.

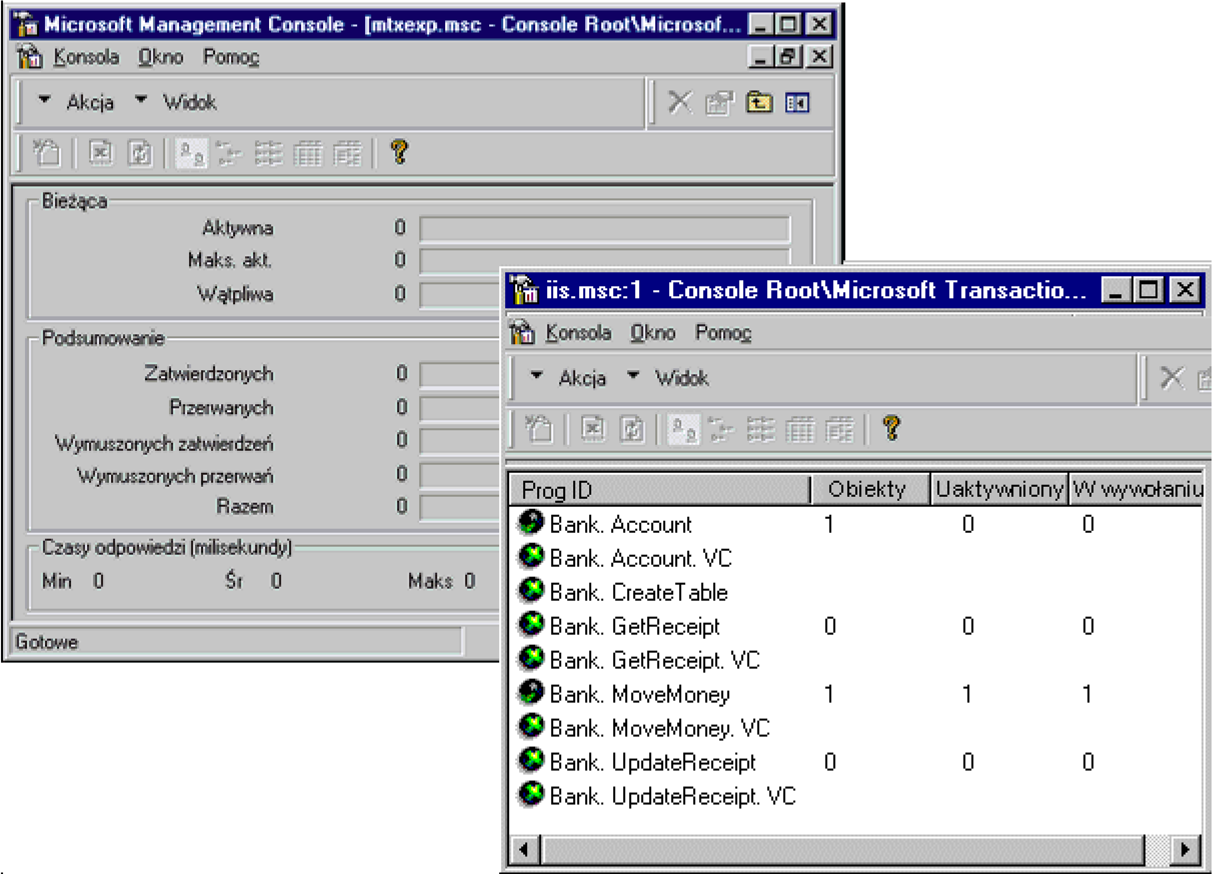
2 W menu **Start** wskaż polecenie **Programy**, wskaż polecenie **Microsoft Transaction Server**, wskaż pozycję **Przykłady**, a następnie kliknij pozycję **Bank Client**. Zmień położenie okna **Bank Client** tak, aby nie zachodziło ono na okna programu MTS Explorer.

3 Domyślnie, na formularzu będzie widoczny kredyt wielkości $1 przydzielony kontu numer 1. Kliknij przycisk **Przekaż**. Na ekranie powinien zostać wyświetlony nowy stan konta.

4 Obserwuj okna programu MTS Explorer. Informacje o wykorzystaniu składników oraz statystyki transakcji powinny być uaktualnione.

5 Eksperymentuj z klientem banku i obserwuj statystyki przy użyciu różnych typów transakcji, serwerów i iteracji. Pierwsza transakcja jest przeprowadzana dłużej od pozostałych z następujących przyczyn:

- Pierwsza transakcja tworzy tabele bazy danych przykładowego banku oraz wstawia tymczasowe rekordy.
- Rozpoczęcie procesu serwera powoduje zużycie zasobów systemu.
- Otwarcie połączeń z bazą danych po raz pierwszy jest kosztowną operacją serwera.



Instalacja przykładowej aplikacji Tic-Tac-Toe programu MTS

Przykładowa aplikacja Tic-Tac-Toe jest grą, w której może uczestniczyć komputer lub inny użytkownik, pracujący na zdalnym komputerze wyposażonym w program MTS. Uruchamiając aplikację Tic-Tac-Toe można z jednej strony przetestować poprawność instalacji programu MTS bez programu SQL Server, z drugiej zaś przećwiczyć operacje rozmieszczania pakietów i administrowania pakietami. Aplikacja Tic-Tac-Toe jest zlokalizowana w podkatalogu \MTS\Samples.

W celu uruchomienia aplikacji przykładowej Tic-Tac-Toe należy:

- Zainstalować program Microsoft Transaction Server.
- Uruchomić klienta Tic-Tac-Toe i rozpocząć grę albo przeciw komputerowi, albo przeciw innemu użytkownikowi.

Aby uruchomić klienta Tic-Tac-Toe

1. Uruchom program tClient.exe, zlokalizowany w głównym katalogu instalacji programu MTS.
2. Wpisz swoją nazwę w polu **Your Name is…**, po czym wybierz opcję gry przeciw komputerowi (Deep Viper) lub przeciw innemu użytkownikowi. Informacje na temat gry z przeciwnikiem zdalnym można uzyskać w temacie Praca ze zdalnymi komputerami w programie MTS.

Po rozpoczęciu gry powróć do okna programu MTS Explorer. Zauważ, że ikona składnika serwera Tic-Tac-Toe wiruje, wskazując, że aplikacja jest aktywna, a instalacja programu MTS jest poprawna. Po zatrzymaniu gry ikona składnika przestanie wirować, ponieważ klient Tic-Tac-Toe nie będzie dłużej korzystać ze składnika serwera Tic-Tac-Toe.

Po kliknięciu polecenia **Stan** z menu **Widok** na ekranie zostaną wyświetlone informacje o wykorzystaniu składnika Tic-Tac-Toe.

Instalacja przykładowych skryptów administracyjnych programu MTS

Przykładowe skrypty administracyjne pokazują, w jaki sposób należy pisać skrypty automatyzujące wybrane procedury programu MTS Explorer. Skrypty takie można pisać w dowolnym języku programowania zgodnym z Automatyzacją, na przykład Visual Basic Scripting Edition (VBScript). Skrypty przykładowe pozwalają zautomatyzować procedurę rozmieszczania aplikacji przykładowej Sample Bank.

W celu uruchomienia skryptów przykładowych należy wcześniej zainstalować program Windows Scripting Host (WSH). Proszę zauważyć, że skrypty te mogą być uruchamiane wyłącznie na platformach systemowych Microsoft Windows NT i Alpha.

Aby zainstalować program Windows Scripting Host, znajdujący się w pakiecie Windows NT 4.0 Option Pack:

1. Jeśli program WSH już został zainstalowany, otwórz menu **Start**, zaznacz opcję **Microsoft Internet Information Server** i kliknij pozycję **Internet Information Server Setup**.
2. W ramach programu instalacyjnego kliknij opcję **Dodaj/Usuń**.
3. Na liście składników odszukaj pozycję **Windows Scripting Host** i zaznacz jej pole wyboru.
4. Kliknij przycisk **Zakończ**. Program instalacyjny pakietu Windows NT 4.0 Option Pack rozpocznie instalację na komputerze programu WSH.

Po zainstalowaniu programu WSH, przykładowe skrypty obiektów administracyjnych są gotowe do użycia. W podkatalogu \MTS\Samples\WSH znajdują się następujące skrypty:

- InstDLL.vbs

  Skrypt powoduje usunięcie istniejących wersji aplikacji Sample Bank, utworzenie nowego pakietu o nazwie Sample Bank, instalację w nowym pakiecie składników z bibliotek DLL aplikacji Sample Bank (napisanych w językach Visual Basic, Visual C++ i Visual J++), zmianę atrybutów transakcji i dodanie nowej roli. Przed uruchomieniem skryptu należy zmodyfikować ścieżkę pliku.

  Poprawne działanie skryptu wymaga, aby rejestr komputera zawierał wszystkie identyfikatory programistyczne składników (progID). Z tego względu przed uruchomieniem skryptu składniki typu Java muszą zostać zarejestrowane za pomocą kreatora Visual Studio 97 ActiveX.
- InstPak.vbs

  Skrypt powoduje zainstalowanie pakietu Sample Bank w środowisku czasu wykonywania programu MTS. Przed uruchomieniem skryptu należy zmodyfikować ścieżkę dostępu.
- Uninst.vbs
- Skrypt powoduje odinstalowanie pakietu Sample Bank ze środowiska czasu wykonywania programu MTS. Przed uruchomieniem skryptu należy zmodyfikować ścieżkę pliku.

  Ponadto, skrypt pozwala uruchomić skrypt InstDll.vbs z okna konsoli, podać ścieżkę dostępu jako parametr usuwania istniejących wersji aplikacji Sample Bank, utworzyć nowy pakiet o nazwie Sample Bank, zainstalować w nowym pakiecie składniki z bibliotek DLL aplikacji Sample Bank (napisanych w językach Visual Basic, Visual C++ i Visual J++), zmienić atrybuty transakcji i dodać nową rolę.
- InstPakCLI.vbs

  Skrypt pozwala uruchomić skrypt InstPak.vbs z okna konsoli oraz zainstalować pakiet Sample Bank w środowisku czasu wykonywania programu MTS, podając jako parametr ścieżkę dostępu.

Więcej informacji na temat skryptowych obiektów administracyjnych oraz przykładowych skryptów można znaleźć w podręczniku *MTS Administrator's Guide*, w temacie Automating MTS Administration (pol. Automatyzacja administrowania w programie MTS).

Ograniczenia przykładowych skryptów administracyjnych

- Przed uruchomieniem skryptu InstDLL.vbs składniki typu Java muszą być zarejestrowane za pomocą kreatora Visual Studio 97 ActiveX .
- Instalacja składników typu Java za pomocą skryptów przykładowych jest obsługiwana tylko w komputerach typu Alpha i i386. Użytkownicy komputerów typu Alpha przed uruchomieniem skryptu InstDLL.vbs powinni usunąć kod importujący składników typu Java.

Wspomaganie użytkownika w programie MTS

Użytkownicy programu MTS mogą uzyskać niezbędną pomoc:

- Korzystając z dokumentacji.
- Odwiedzając stronę główną programu Microsoft Transaction Server pod adresem http://www.microsoft.com/transaction/.
- Kontaktując się z firmami oferującymi pomoc techniczną dla programu MTS.

Korzystanie z dokumentacji

Dokumentacja programu MTS zawiera informacje o sposobie działania, możliwych zastosowaniach i używaniu różnych funkcji programu MTS. Podczas korzystania z programu MTS Explorer i słów kluczowych języków programowania można uzyskać dostęp do Pomocy kontekstowej, naciskając klawisz F1. *Pomoc kontekstowa* pozwala dotrzeć do niezbędnych informacji bezpośrednio, bez konieczności wywoływania menu **Pomoc** — wystarczy po prostu nacisnąć klawisz F1.

Zobacz też Omówienie dokumentacji programu MTS

Serwis programu

Firma Microsoft oferuje klientom wiele różnych możliwości wpomagania, dzięki czemu program MTS może być wykorzystany w sposób najbardziej efektywny.

W razie jakichkolwiek pytań dotyczących programu, najpierw należy sięgnąć do Pomocy. Jeśli wątpliwości pozostaną, należy skontaktować się z serwisem firmy Microsoft.

Wprowadzenie do podrêcznika MTS Administrator’s Guide

Podrêcznik MTS Administrator’s Guide zawiera informacje na temat zastosowania programu Microsoft® Transaction Server Explorer do tworzenia, instalowania, rozpowszechniania i utrzymywania ¿¹danego stanu pakietów. Podrêcznik ten jest przeznaczony dla nastêpuj¹cych grup u¿ytkowników:

- Administratorzy systemu
- Administratorzy sieci Web
- Projektanci aplikacji

Projektanci mog¹ odwo³ywaæ siê do procedur opisanych w podrêczniku podczas tworzenia, rozmieszczania i rozpowszechniania aplikacji programu MTS. Administratorzy systemów i sieci Web mog¹ korzystaæ z okreœlonych procedur lub specyficznych zadañ rozmieszczaj¹c, administruj¹c i utrzymuj¹c ¿¹dany stan aplikacji programu MTS.

W wymienionych ni¿ej rozdzia³ach wyjaœniono procedury rozmieszczania pakietów i administrowania pakietami za pomoc¹ programu MTS Explorer oraz umieszczono ³¹cza do bardziej szczegó³owych tematów:

Na czym polega tworzenie pakietów programu MTS?

Na czym polega rozpowszechnianie pakietów programu MTS?

Na czym polega instalacja pakietów programu MTS?

Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?

Na czym polega zarz¹dzanie transakcjami programu MTS?

Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Program MTS Explorer w systemie Windows 95

Program MTS Explorer mo¿e byæ u¿ywany do zarz¹dzania pakietami programu MTS w systemie operacyjnym Windows® 95. Tym niemniej, administrowanie w systemie Windows 95 ma nastêpuj¹ce ograniczenia:

- Komputer wyposa¿ony w system Windows NT mo¿na wykorzystaæ do zdalnego administrowania komputerem wyposa¿onym w system Windows 95 pod warunkiem, ¿e na tym drugim zainstalowano us³ugê Rejestr zdalny (ang. Remote Registry). Us³uga ta pozwala zmieniaæ wpisy do rejestru w zdalnych komputerach wyposa¿onych w system Windows 95 (jeœli u¿ytkownik ma odpowiednie uprawnienia).

  Aby uzyskaæ us³ugê Rejestr zdalny, nale¿y skorzystaæ z dysku CD-ROM systemu Windows 95 i przejœæ do katalogu \Admin\Nettols\Remotereg. Tutaj nale¿y zapoznaæ siê z treœci¹ pliku Regserv.txt zawieraj¹cego informacje na temat instalowania us³ugi Rejestr zdalny, a nastêpnie uruchomiæ program instalacyjny us³ugi (Regserv.exe).
- Zdalne administrowanie programem MTS uruchomionym w systemie Windows 95 jest niemo¿liwe, niezale¿nie od tego, czy odbywa siê to za pomoc¹ komputera wyposa¿onego w system Windows 95 czy Windows NT.
- Nie jest wyœwietlane lewe okienko programu MTS Explorer, zawieraj¹ce drzewo katalogów. Aby przejœæ w dó³ drzewa, nale¿y klikn¹æ dwukrotnie wybran¹ ikonê, aby przejœæ w górê drzewa, nale¿y skorzystaæ z przycisku **Poziom wy¿ej** (wyœwietlanego na pasku narzêdzi).
- Poniewa¿ w systemie Windows 95 nie jest obs³ugiwane narzêdzie do tworzenia plików wykonywalnych aplikacji, programu MTS Explorer nie mo¿na wykorzystaæ do generowania plików wykonywalnych aplikacji. Wiêcej informacji na temat narzêdzia do tworzenia plików wykonywalnych aplikacji mo¿na uzyskaæ pod has³em Generowanie plików wykonywalnych programu MTS.

- W systemie Windows 95 nie s¹ obs³ugiwane role i w³aœciwoœci zabezpieczeñ programu MTS. Z tego wzglêdu nie mo¿na w nim przegl¹daæ w Eksploratorze folderów "Role", "Przynale¿noœæ ról" i "U¿ytkownicy".
- Sk³adniki uruchamiane w systemie Windows 95 s¹ niedostêpne dla klientów zdalnych pracuj¹cych na innych komputerach.
- Poniewa¿ w systemie Windows 95 nie wystêpuje systemowy dziennik zdarzeñ, wpisy do dziennika zdarzeñ s¹ zapisywane w pliku HTML o nazwie Transaction Server.html, który znajduje siê w katalogu Windows, w podkatalogu \MTSLogs. Plik ten mo¿na wykorzystaæ do monitorowania interesuj¹cych zdarzeñ w systemie lub programie.

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server

Na czym polega tworzenie pakietów pogramu MTS?

Tworzenie pakietów jest ostatnim elementem procesu projektowania aplikacji. Projektanci pakietów i administratorzy sieci Web wykorzystuj¹ program MTS Explorer do tworzenia i rozmieszczania pakietów. Zwyk³y u¿ytkownik korzysta zeñ w celu zaimplementowania konfiguracji pakietu i sk³adników, okreœlonej przez projektanta aplikacji.

Wiêcej informacji na temat projektowania i budowania aplikacji programu MTS mo¿na znaleŸæ w ksi¹¿ce *MTS Programmer's Guide*, w rozdziale Budowanie aplikacji programu MTS.

Procedury:

Tworzenie pustego pakietu programu MTS

Dodawanie sk³adnika do pakietu programu MTS

Import sk³adnika do pakietu programu MTS

Usuwanie sk³adnika z pakietu programu MTS

Budowanie pakietu programu MTS w celu eksportu

Ustawianie w³aœciwoœci pakietu programu MTS

Ustawianie w³aœciwoœci aktywacji programu MTS

Ustawianie w³aœciwoœci transakcji programu MTS

Ustawianie poziomów uwierzytelnienia programu MTS

Blokowanie pakietu programu MTS

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega rozpowszechnianie pakietów programu MTS?, Na czym polega instalacja pakietów programu MTS?, Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?, Na czym polega zarz¹dzanie transakcjami programu MTS?, Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Na czym polega rozpowszechnianie pakietów programu MTS?

Po zbudowaniu niezbêdnych sk³adników aplikacji programu MTS i umieszczeniu ich w pakiecie mo¿na przyst¹piæ do rozpowszechniania aplikacji wœród klientów. Aplikacje mo¿na rozpowszechniaæ na dwa sposoby:

- Przesy³aæ sk³adniki z w³asnego serwera do serwera administratora systemu lub sieci Web wyposa¿onego w program MTS Explorer. W takim wypadku obydwa komputery musz¹ korzystaæ z programu MTS.
- Za pomoc¹ narzêdzia do tworzenia plików wykonywalnych aplikacji (programu MTS Explorer) automatycznie generowaæ pliki wykonywalne aplikacji, do których kienci mogliby siê odwo³ywaæ ze swoich serwerów zdalnych. Aplikacja klienta nie musi wówczas korzystaæ z programu MTS.

Rozpowszechnianie pakietów przy u¿yciu narzêdzia do tworzenia plików wykonywalnych aplikacji (2. sposób) jest zalecane w odniesieniu do klientów, którzy nie korzystaj¹ z programu MTS. Plik wykonywalny aplikacji automatycznie konfiguruje komputery klientów tak, aby zapewniæ im dostêp do sk³adników uruchamianych na serwerze zdalnym programu MTS. Za pomoc¹ programu MTS Explorer sk³adniki zdalne mo¿na tak¿e konfigurowaæ samodzielnie (rêcznie).

Mimo i¿ stosowanie narzêdzia do tworzenia plików wykonywalnych aplikacji nie wymaga wiedzy programistycznej, dystrybutorzy aplikacji programu MTS powinni byæ dok³adnie zaznajomieni z konsekwencjami decyzji podejmowanych podczas tworzenia pakietów i rozpowszechniania aplikacji (serwerowych i klienckich). Na przyk³ad utworzenie nieprawid³owego pakietu aplikacji mo¿e spowodowaæ pojawienie siê w pliku wykonywalnym klienta kodu aplikacji serwera.

Procedury:

Praca ze zdalnymi komputerami programu MTS

Eksport pakietów programu MTS

Generowanie plików wykonywalnych programu MTS

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega tworzenie pakietów programu MTS?, Na czym polega instalacja pakietów programu MTS?, Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?, Na czym polega zarz¹dzanie transakcjami programu MTS?, Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Na czym polega instalacja pakietów programu MTS?

Po zbudowaniu pakietu nale¿y go zainstalowaæ i rozmieœciæ, co wymaga znajomoœci ró¿nych w³aœciwoœci pakietu i jego sk³adników. Na przyk³ad po zainstalowaniu pakietu administrator systemu lub sieci Web musi przypisaæ (zamapowaæ) u¿ytkowników systemu Windows NT do skojarzonych z pakietem ról. Aby zrobiæ to w³aœciwie, musi znaæ dok³adnie znaczenie poszczególnych ról oraz mieæ wiedzê na temat zabezpieczeñ deklaratywnych opartych na rolach.

Procedury:

Instalacja pakietów wstêpnie wbudowanych

Uaktualnianie pakietów programu MTS

W³¹czanie zabezpieczeñ pakietu programu MTS

Ustawianie to¿samoœci pakietu programu MTS

Dodawanie nowej roli programu MTS

Mapowanie ról programu MTS do u¿ytkowników i grup

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega tworzenie pakietów programu MTS?, Na czym polega rozpowszechnianie pakietów programu MTS?, Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?, Na czym polega zarz¹dzanie transakcjami programu MTS?, Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?

Program MTS Explorer mo¿na wykorzystaæ do utrzymywania ¿¹danego stanu aplikacji programu MTS. Umo¿liwiaj¹ to mechanizmy monitorowania stanu zainstalowanych pakietów oraz ponownej konfiguracji w³aœciwoœci pakietów i sk³adników (jeœli jest to mo¿liwe). W niniejszym podrozdziale opisano metody ponownej konfiguracji pakietów, które ju¿ zainstalowano i rozmieszczono.

Procedury:

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Korzystanie z arkuszy w³aœciwoœci w programie MTS Explorer

Zarz¹dzanie u¿ytkownikami dla ról programu MTS

Korzystanie z replikacji programu MTS

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega tworzenie pakietów programu MTS?, Na czym polega rozpowszechnianie pakietów programu MTS?, Na czym polega instalacja pakietów programu MTS?, Na czym polega zarz¹dzanie transakcjami programu MTS?, Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Na czym polega zarz¹dzanie transakcjami programu MTS?

Poprawne zarz¹dzanie transakcjami za pomoc¹ programu MTS Explorer wymaga dok³adnego zrozumienia zasad dzia³ania transakcji. Niniejszy podrozdzia³ zawiera podstawowe informacje na temat transakcji i stanów transakcji, a tak¿e metod monitorowania i zarz¹dzania transakcjami przez administratorów.

Procedury:

Podstawowe informacje o transakcjach programu MTS

Zarz¹dzanie us³ug¹ MS DTC

Monitorowanie transakcji programu MTS

Monitorowanie transakcji programu MTS w systemie Windows 95

Podstawowe informacje o stanach transakcji programu MTS

Przeprowadzanie transakcji programu MTS

Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega tworzenie pakietów programu MTS?, Na czym polega rozpowszechnianie pakietów programu MTS?, Na czym polega instalacja pakietów programu MTS?, Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?, Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Na czym polega automatyzacja administrowania za pomoc¹ programu MTS?

Automatyzacja konfigurowania pakietów i sk³adników w programie MTS Explorer jest mo¿liwa dziêki skryptom. Aby automatyzowaæ ró¿nego rodzaju czynnoœci administracyjne programu MTS Explorer, wystarczy znaæ jakikolwiek jêzyk do tworzenia skryptów, na przyk³ad Microsoft Visual Basic. Programiœci bardzo czêsto wykorzystuj¹ obiekty skryptowe do tworzenia dodatkowych narzêdzi programu MTS Explorer, tzw. dodatków (ang. add-ins), na przyk³ad obiektów powoduj¹cych automatyczn¹ konfiguracjê klientów zdalnych.

Uwaga Korzystanie z obiektów skryptowych wymaga znajomoœci jednego z jêzyków do tworzenia skryptów, zgodnego z Automatyzacj¹.

Aby mieæ dostêp do przyk³adowych skryptów administracyjnych oraz podrêcznika *MTS Administrative Reference*, nale¿y zainstalowaæ przyk³adowe projekty i dokumentacjê programu MTS. Ksi¹¿ka *MTS Administrative Reference* zawiera strony odwo³añ API oraz przyk³ady kodu skryptów napisanych w jêzyku Microsoft Visual Basic i Microsoft Visual C++® API. Przyk³adowe skrypty administracyjne s¹ napisane w jêzyku Visual Basic Script i korzystaj¹ z funkcji programu Windows Scripting Host, który mo¿na zainstalowaæ za pomoc¹ programu instalacyjnego pakietu Windows NT 4.0 Option Pack. Wiêcej informacji na temat korzystania ze skryptów administracyjnych programu MTS mo¿na znaleŸæ pod has³em Instalacja przyk³adowych skryptów administracyjnych programu MTS.

Procedury:

Obiekty administracyjne programu MTS

Przyk³ady skryptów napisanych w jêzyku Visual Basic, automatyzuj¹cych administrowanie w programie MTS

Przyk³adowa aplikacja utworzona za pomoc¹ programu Visual Basic, automatyzuj¹ca administrowanie w programie MTS

Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Zobacz te¿

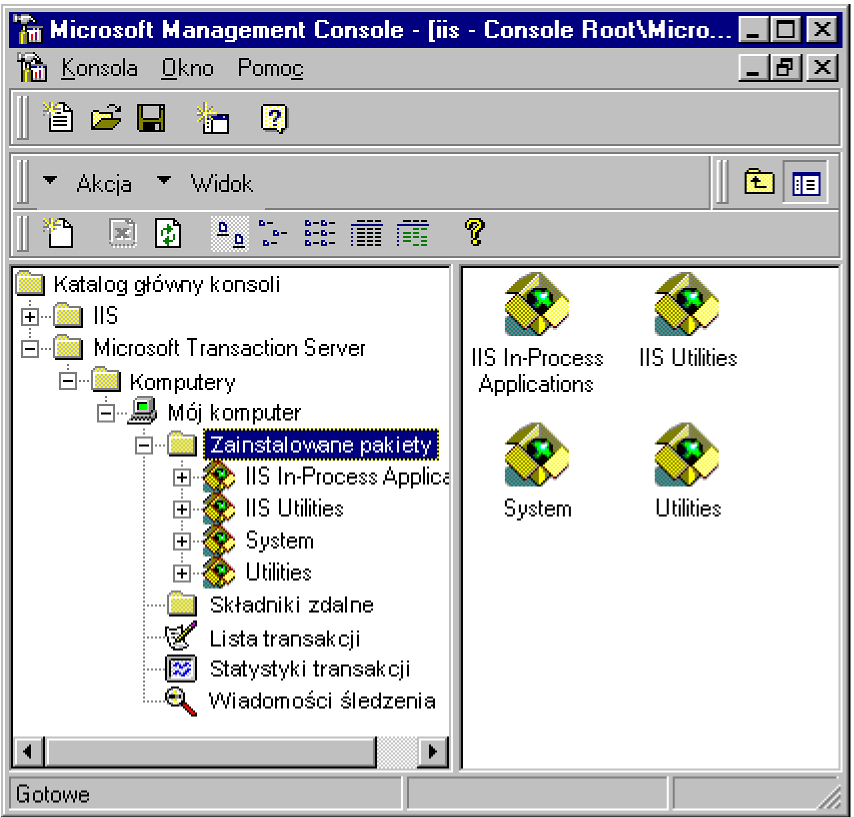
Krótkie wprowadzenie do programu Microsoft Transaction Server, Rozpoczêcie pracy z programem Microsoft Transaction Server , Na czym polega tworzenie pakietów programu MTS?, Na czym polega instalacja pakietów programu MTS?, Na czym polega utrzymanie ¿¹danego stanu pakietów programu MTS?, Na czym polega zarz¹dzanie transakcjami programu MTS?

Hierarchia programu MTS Explorer

Program MTS Explorer jest narzêdziem (wizualnym) do zarz¹dzania pakietami i sk³adnikami programu MTS, dzia³aj¹cym w œrodowisku czasu wykonywania programu MTS. Za pomoc¹ programu MTS Explorer mo¿na wykonywaæ ró¿nego rodzaju zadania, od instalowania sk³adników w pakietach do monitorowania stanu transakcji.

Program MTS Explorer jest przystawk¹ zarz¹dzan¹ przez program Microsoft Management Console (MMC) w systemie Windows NT oraz przez plik wykonywalny (mtxpd.exe) w systemie Windows 95. W lewym okienku programu MTS Explorer jest wyœwietlane drzewo hierarchii programu, zawieraj¹ce nastêpuj¹ce foldery:

- **Komputery**
  Folder zawiera komputery zarz¹dzane z bie¿¹cego serwera u¿ytkownika. Komputer lokalny ma nazwê "Mój Komputer".
- **Zainstalowane pakiety**
  Folder zawiera pakiety zainstalowane na danym komputerze.
- **Sk³adniki zdalne**
  Folder zawiera sk³adniki z komputerów zdalnych wykorzystywane przez pakiety danego komputera.
- **Lista transakcji**
  Umo¿liwia obejrzenie wszystkich bie¿¹cych transakcji, w których uczestniczy aplikacja programu MTS.
- **Statystyki transakcji**
  Umo¿liwia obejrzenie statystyk transakcji, w których uczestniczy komputer lokalny.
- **Komunikaty œledzenia**
  Pozwala obejrzeæ listê bie¿¹cych komunikatów œledzenia wysy³anych przez us³ugê Microsoft Distributed Transaction Coordinator (MS DTC).



Uwaga W systemie Windows 95 program MTS Explorer ma jedno okienko. Aby poruszaæ siê w dó³ drzewa (hierarchii) programu, nale¿y klikn¹æ dwukrotnie wybran¹ ikonê, aby poruszaæ siê w górê drzewa mo¿na klikn¹æ przycisk **W górê o jeden poziom**, wyœwietlany na pasku narzêdzi.

Ponadto, w systemie Windows 95 nie mo¿na korzystaæ z folderu "Sk³adniki zdalne" ani z narzêdzia do tworzenia plików wykonywalnych aplikacji.

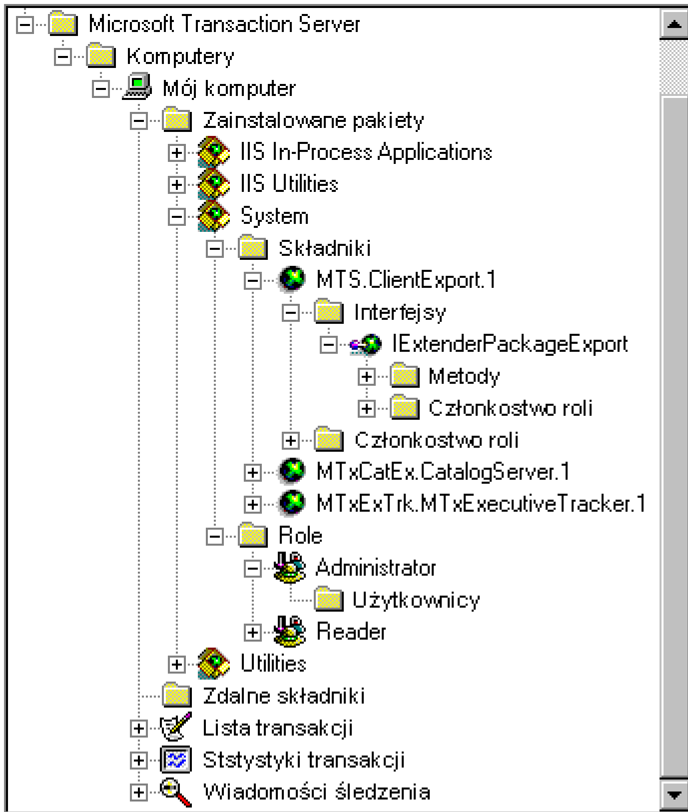
Wiêcej informacji na temat ograniczeñ zwi¹zanych z prac¹ programu MTS Explorer w systemie Windows 95 mo¿na znaleŸæ pod has³em Wprowadzenie do podrêcznika MTS Administrator's Guide, w podrozdziale *Program MTS Explorer w systemie Windows 95*.

W komputerach zdalnych, administrowanych za pomoc¹ programu MTS Explorer, okienka **Zainstalowane pakiety**, **Sk³adniki zdalne**, **Lista transakcji**, **Statystyki transakcji** i **Komunikaty œledzenia** s¹ wyœwietlane w drzewie pod ikon¹ komputera.

Wszystkie pakiety zainstalowane w programie MTS Explorer zawieraj¹ nastêpuj¹ce podfoldery:

- **Sk³adniki**
  Zawiera sk³adniki zainstalowane w wybranym pakiecie .
- **Role**
  Zawiera dostêpne w danym pakiecie role.
- **Przynale¿noœæ ról**
  Zawiera role przypisane do wybranego sk³adnika lub interfejsu sk³adnika.
- **U¿ytkownicy**
  Zawiera u¿ytkowników przypisanych do roli pakietu.
- **Interfejsy**
  Zawiera interfejsy dla wybranego sk³adnika.

- **Metody**
  Zawiera metody dla wybranego interfejsu.



Zobacz te¿

Krótkie wprowadzenie do programu Microsoft Transaction Server

Folder "Komputery"

Folder "Komputery" zawiera komputer oznaczony jako "Mój komputer" oraz wszystkie inne komputery dodane doñ przez u¿ytkownika. Domyœlnie nazwa "Mój komputer" odnosi siê do komputera lokalnego, na którym zainstalowano program MTS.

Aby do folderu "Komputery" dodaæ nowy komputer, mo¿na wykonaæ jedn¹ z nastêpuj¹cych czynnoœci:

- Klikn¹æ folder "Komputery" prawym przyciskiem myszy, a nastêpnie wybraæ polecenia **Nowy** i **Komputer**.
- Zaznaczyæ folder "Komputery" i klikn¹æ przycisk ikonê **Utwórz nowy obiekt**, znajduj¹c¹ siê na pasku narzêdzi w prawym okienku programu.
- Zaznaczyæ folder "Komputery", w lewym okienku programu MTS Explorer otworzyæ menu **Akcja** i wybraæ z niego polecenie **Nowy**.

W okienku dialogowym **Dodaj komputer** nale¿y wpisaæ nazwê serwera, który bêdzie administrowany z bie¿¹cego komputera. Nowy serwer zostanie dodany do folderu "Komputery". W hierarchii programu MTS Explorer folder ten znajduje siê pod ikon¹ **Mój Komputer**.

Zobacz te¿

Mój komputer**,** W³aœciwoœci komputera

Mój komputer

Nazwa "Mój komputer" odnosi siê do komputera lokalnego, w którym zainstalowano program MTS.

Procedury wykonywane na poziomie folderu "Komputer" opisano w nastêpuj¹cych tematach Pomocy:

Korzystanie z replikacji programu MTS

Zarz¹dzanie us³ug¹ MS DTC

Wszystkie foldery "Komputer" zawieraj¹ nastêpuj¹ce foldery:

"Zainstalowane pakiety"

Folder zawiera pakiety systemowe i zainstalowane przez u¿ytkownika, zarz¹dzane z tego komputera.

"Sk³adniki zdalne"

Folder zawiera sk³adniki z komputerów zdalnych wywo³ywane przez pakiety komputera lokalnego.

"Lista transakcji"

Umo¿liwia obejrzenie listy bie¿¹cych transakcji zarz¹dzanych przez ten komputer.

"Statystyki transakcji"

Umo¿liwia obejrzenie statystyk transakcji, w których uczestniczy ten komputer.

"Komunikaty œledzenia"

Pozwala obejrzeæ listê bie¿¹cych komunikatów œledzenia wysy³anych przez us³ugê Microsoft Distributed Transaction Coordinator (MS DTC).

W³aœciwoœci komputera oznaczonego jako "Mój komputer", podobnie jak w³aœciwoœci wszystkich innych komputerów z folderu "Komputery", mo¿na konfigurowaæ za pomoc¹ nastêpuj¹cych arkuszy w³aœciwoœci:

- **Ogólne**
- **Opcje**
- **Zaawansowane**

Zobacz te¿

Folder "Komputery"

Folder "Zainstalowane pakiety"

Folder "Zainstalowane pakiety" zawiera listê wszystkich pakietów dodanych do danego komputera. Na poziomie pakietu mo¿na wykonywaæ nastêpuj¹ce czynnoœci:

Tworzenie pustego pakietu programu MTS

Instalacja wstêpnie wbudowanych pakietów programu MTS

Uaktualnianie pakietów programu MTS

Ustawianie w³aœciwoœci pakietów programu MTS

Ustawianie w³aœciwoœci aktywacji programu MTS

Ustawianie to¿samoœci pakietu programu MTS

W³¹czanie zabezpieczeñ pakietów programu MTS

Ustawianie w³aœciwoœci transakcji programu MTS

Blokowanie pakietów programu MTS

Eksport pakietów programu MTS

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Korzystanie z arkuszy w³aœciwoœci w programie MTS Explorer

Folder "Zainstalowane pakiety" zawiera równie¿ nastêpuj¹ce pakiety systemowe:

"System"

"Narzêdzia"

Uwaga S¹ to pakiety wewnêtrzne programu MTS, w zwi¹zku z czym u¿ytkownicy nie powinni ich zmieniaæ ani konfigurowaæ. Niemniej jednak, ograniczenie lub rozszerzenie uprawnieñ serwera programu MTS mo¿e wymagaæ zmiany konfiguracji wewnêtrznych pakietów programu MTS. Na przyk³ad aby przydzieliæ danemu u¿ytkownikowi serwera programu MTS uprawnienia administracyjne, nale¿y dodaæ tego u¿ytkownika do roli Administrator pakietu "System".

W folderze "Zainstalowane pakiety" mo¿na zainstalowaæ dowoln¹ liczbê pakietów. Ka¿dy zainstalowany pakiet zawiera nastêpuj¹ce podfoldery.

- "Sk³adniki"
  Zawiera sk³adniki zainstalowane w wybranym pakiecie
- "Role"
  Zawiera role przypisane do wybranego pakietu.

W³aœciwoœci pakietów mo¿na konfigurowaæ za pomoc¹ nastêpuj¹cych arkuszy w³aœciwoœci:

- **Ogólne**
- **Zabezpieczenia**
- **Zaawansowane**
- **To¿samoœæ**
- **Aktywacja**

Jeœli program MTS wspó³pracuje z programem Internet Information Server (IIS) w wersji 4.0, folder "Zainstalowane pakiety" zawiera równie¿ nastêpuj¹ce pakiety systemowe zwi¹zane z programem IIS:

- "IIS In-Process Applications"
  Folder "IIS In-Process Applications" zawiera sk³adniki dla wszystkich aplikacji programu Internet Information Server (IIS) uruchomionych w procesie tego programu. Aplikacja

programu IIS mo¿e dzia³aæ w procesie programu IIS lub w oddzielnym procesie aplikacji. Jeœli aplikacja taka dzia³a w procesie programu IIS, jest umieszczana w folderze "IIS In-Process Applications" jako sk³adnik. Jeœli aplikacja taka dzia³a w oddzielnym procesie, jest traktowana inaczej i przedstawiana w hierarchii programu MTS Explorer jako osobny pakiet.

- "IIS Utilities"

  Folder "IIS Utilities" zawiera obiekt ObjectContext, niezbêdny do w³¹czania transakcji w œrodowisku Active Server Pages (ASPs). Wiêcej informacji na temat transakcyjnych stron ASP mo¿na znaleŸæ w dokumentacji programu Internet Information Server (IIS).

  Program Microsoft Transaction Server wykorzystuje sk³adnik ObjectContext dla potrzeb funkcji wewnêtrznych. W³aœciwoœci tego sk³adnika mo¿na przegl¹daæ, ale nie mo¿na ich ustawiaæ.

Pakiet "Utilities"

Pakiet "Utilities" zawiera dwa sk³adniki: **TransactionContext** i **TransactionContextEx**. Sk³adnik **TransactionContext/TransactionContextEx** mo¿na wykorzystaæ w bazowych aplikacjach klienckich do po³¹czenia efektów dzia³ania kilku obiektów programu Microsoft Transaction Server w jedn¹ transakcjê atomow¹, a tak¿e do przerywania i przekazywania transakcji.

Zobacz te¿

Zarz¹dzanie transakcjami programu MTS

Pakiet "System"

Pakiet "System" zawiera sk³adniki, które nie mog¹ byæ modyfikowane. Sk³adniki te s¹ wykorzystywane w programie Microsoft Transaction Server dla potrzeb funkcji wewnêtrznych. W³aœciwoœci pakietu "System" mo¿na przegl¹daæ, ale nie mo¿na ich ustawiaæ.

Aby u¿ytkownik móg³ usuwaæ i modyfikowaæ pakiety zarz¹dzane przez program MTS Explorer, nale¿y przypisaæ go do roli Administrator pakietu "System". Jeœli program MTS zainstalowano na serwerze, którego rolê ustawiono jako podstawowy lub zapasowy kontroler domeny, wówczas uprawnienia zarz¹dzania pakietami za pomoc¹ programu MTS Explorer maj¹ tylko administratorzy domeny.

Zobacz te¿

Konfiguracja serwera programu MTS, W³¹czanie zabezpieczeñ pakietów programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup, Folder "U¿ytkownicy"

Folder "Sk³adniki"

Folder "Sk³adniki" zawiera sk³adniki wybranego pakietu.

Procedury wykonywane na poziomie folderu sk³adników opisano w nastêpuj¹cych tematach Pomocy:

Dodawanie sk³adnika do pakietu programu MTS

Import sk³adnika do pakietu programu MTS

Usuwanie sk³adnika z pakietu

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Korzystanie z arkuszy w³aœciwoœci w programie MTS Explorer

Ustawianie w³aœciwoœci transakcji programu MTS

Konfiguracja ról MTS dla potrzeb zabezpieczeñ deklaratywnych

Folder "Sk³adniki" zawiera nastêpuj¹ce podfoldery:

- "Interfejsy"
  Zawiera interfejsy skojarzone z wybranym sk³adnikiem.
- "Przynale¿noœæ ról (Sk³adniki)"
  Zawiera role i u¿ytkowników skojarzonych z rolami wybranego sk³adnika.

W³aœciwoœci sk³adników mo¿na konfigurowaæ za pomoc¹ nastêpuj¹cych arkuszy w³aœciwoœci:

- **Ogólne**
- **Transakcja**
- **Zabezpieczenia**

Zobacz te¿

Ustawianie w³aœciwoœci aktywacji programu MTS

Folder "Role"

Folder "Role" zawiera role przypisane wybranemu pakietowi. Definiowane w programie MTS role okreœlaj¹ zakres dostêpu u¿ytkowników do pakietów, sk³adników i interfejsów. Ka¿da rola dodana do folderu "Role" wybranego pakietu mo¿e zostaæ dodana równie¿ do folderu "Przynale¿noœæ ról" sk³adnika lub interfejsu tego pakietu.

Na poziomie folderu "Role" mo¿na wykonywaæ nastêpuj¹ce czynnoœci:

Mapowanie ról programu MTS do u¿ytkowników i grup

W³¹czanie zabezpieczeñ pakietów programu MTS

Dodawanie nowej roli programu MTS

Folder "Role" zawiera jeden podfolder:

- "U¿ytkownicy"

W³aœciwoœci ról mo¿na konfigurowaæ za pomoc¹ nastêpuj¹cego arkusza w³aœciwoœci:

- **Ogólne**

Zobacz te¿

Konfiguracja serwera programu MTS, Pakiet "System"

Folder "Interfejsy"

Folder "Interfejsy" zawiera interfejsy zdefiniowane dla wybranego sk³adnika.

Folder "Interfejsy" zawiera dwa podfoldery:

- "Metody"
- "Przynale¿noœæ ról"

W³aœciwoœci interfejsów mo¿na przegl¹daæ za pomoc¹ nastêpuj¹cych arkuszy w³aœciwoœci:

- **Ogólne**
- **Proxy**

W³aœciwoœci interfejsu nie mo¿na konfigurowaæ bez podawania jego opisu na arkuszu w³aœciwoœci **Ogólne**.

Zobacz te¿

W³¹czanie zabezpieczeñ pakietu programu MTS

Folder "Metody"

Folder "Metody" zawiera metody zdefiniowane dla wybranego interfejsu.

W³aœciwoœci metod mo¿na ustawiaæ za pomoc¹ jednego arkusza w³aœciwoœci:

- **Ogólne**

Na arkuszu w³aœciwoœci **Ogólne** metody s¹ konfigurowane poprzez dodanie ich opisu.

Zobacz te¿

Folder "Interfejsy'

Folder "Przynale¿noœæ ról"

Folder "Przynale¿noœæ ról" zawiera role skojarzone ze sk³adnikiem lub interfejsem. Jeœli role te zostan¹ dodane do folderu "Przynale¿noœæ ról" z folderu "Role" pakietu, wówczas mo¿na bêdzie sprawdzaæ, komu udostêpniany jest interfejs lub sk³adnik.

W³aœciwoœci przynale¿noœci ról ustawia siê na poziomie folderu "Role". Na przyk³ad opis okreœlony dla roli pakietu jest wyœwietlany na poziomie folderu "Przynale¿noœæ ról".

Wiêcej informacji na temat zabezpieczeñ deklaratywnych mo¿na znaleŸæ pod has³em W³¹czanie zabezpieczeñ pakietu programu MTS.

Zobacz te¿

Folder "Interfejsy"

Folder "U¿ytkownicy"

Folder "U¿ytkownicy" zawiera grupy i u¿ytkowników systemu Windows NT, których skojarzono z dan¹ rol¹. Ka¿dy u¿ytkownik reprezentuje konto u¿ytkownika systemu Windows NT dodane do folderu "Role" pakietu. Dodaj¹c konta u¿ytkowników lub grupy systemu Windows NT do folderu "Role", mo¿na kontrolowaæ dostêp do sk³adników, pakietów i interfejsów.

Na poziomie folderu "U¿ytkownicy" mo¿na wykonywaæ nastêpuj¹ce czynnoœci:

Dodawanie nowego u¿ytkownika do roli
Usuwanie u¿ytkownika z roli

Z folderem "U¿ytkownicy" nie skojarzono ¿adnych arkuszy w³aœciwoœci.

Zobacz te¿

W³¹czanie zabezpieczeñ pakietu programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup

Folder "Sk³adniki zdalne"

Folder "Sk³adniki zdalne" zawiera sk³adniki, które zarejestrowano lokalnie, na tym komputerze, w celu ich zdalnego uruchamiania na innym komputerze. Korzystanie z folderu "Sk³adniki zdalne" wymaga zainstalowania programu MTS na konfigurowanych komputerach klientów. Jeœli komputery zdalne maj¹ byæ konfigurowane rêcznie, za pomoc¹ programu MTS Explorer, do folderu "Sk³adniki zdalne" nale¿y dodaæ wszystkie sk³adniki, do których komputery zdalne bêd¹ uzyskiwaæ dostêp.

Przed rozpoczêciem konfiguracji sk³adników zdalnych, do folderu "Komputery" nale¿y dodaæ wszystkie dodatkowe serwery, które bêd¹ uruchamiaæ sk³adniki zdalne.

Aby uzyskaæ wiêcej informacji o pracy z komputerami zdalnymi korzystaj¹cymi z programu MTS, mo¿na siêgn¹æ do nastêpuj¹cych tematów Pomocy:

Eksport pakietów programu MTS

Praca z komputerami zdalnymi korzystaj¹cymi z programu MTS

Uwaga Sk³adniki dzia³aj¹ce w systemie Windows 95 nie mog¹ byæ u¿ywane zdalnie przez klientów pracuj¹cych na innych komputerach. Wiêcej informacji na ten temat mo¿na uzyskaæ pod has³em Wprowadzenie do podrêcznika MTS Administrator's Guide, w podrozdziale *Program MTS Explorer w systemie Windows 95*

Zobacz te¿

Generowanie plików wykonywalnych programu MTS

Lista transakcji

W oknie **Lista transakcji** s¹ wyœwietlane bie¿¹ce transakcje, w których uczestniczy komputer. W tym:

- Transakcje o stanie "W¹tpliwa".
- Transakcje, które pozostawa³y w tym samym stanie przez okres czasu ustalony na arkuszach w³aœciwoœci komputera, na karcie **Zaawansowane**.

Opis ikon wyœwietlanych w oknie **Lista transakcji** mo¿na znaleŸæ pod has³em Ikony transakcji.

Zagadnienia zwi¹zane z zarz¹dzaniem transakcjami opisano w nastêpuj¹cych tematach:

- Podstawowe informacje o transakcjach programu MTS
- Podstawowe informacje o stanach transakcji programu MTS

Za pomoc¹ okna **Lista transakcji** mo¿na wykonywaæ nastêpuj¹ce zadania:

- Monitorowanie transakcji programu MTS
- Monitorowanie transakcji programu MTS w systemie Windows 95
- Przeprowadzanie transakcji programu MTS

W³aœciwoœci transakcji mo¿na obejrzeæ, klikaj¹c wybran¹ transakcjê prawym przyciskiem myszy i wybieraj¹c z menu podrêcznego polecenie **W³aœciwoœci**.

Zobacz te¿

Ikony transakcji, Zarz¹dzanie us³ug¹ MS DTC

Statystyki transakcji

W oknie **Statystyki transakcji** s¹ wyœwietlane statystyki transakcji, w których uczestniczy komputer. Niektóre statystyki maj¹ charakter zbiorczy; inne pokazuj¹ bie¿¹cy stan dzia³añ.

Zagadnienia zwi¹zane z zarz¹dzaniem transakcjami opisano w nastêpuj¹cych tematach:

- Podstawowe informacje o transakcjach programu MTS
- Podstawowe informacje o stanach transakcji programu MTS

Za pomoc¹ okna **Statystyki transakcji** mo¿na wykonywaæ nastêpuj¹ce zadania:

- Monitorowanie transakcji programu MTS
- Monitorowanie transakcji programu MTS w systemie Windows 95
- Przeprowadzanie transakcji programu MTS

Jeœli program MTS Explorer jest u¿ywany w systemie Windows NT, okno **Statystyki transakcji** mo¿na otwieraæ tylko dla danego serwera.

Bie¿¹ca

- **Aktywna** — Bie¿¹ca liczba transkacji, dla których nie zosta³ jeszcze zakoñczony protokó³ przekazywania dwufazowego.
- **Maks. aktywna** — Najwiêksza liczba transakcji aktywnych podczas bie¿¹cej sesji us³ugi Microsoft Distributed Transaction Coordinator (MS DTC).
- **W¹tpliwa** — Aktualna liczba transakcji, których nie mo¿na przekazaæ z powodu awarii komunikacji miêdzy serwerem lokalnej bazy danych a koordynatorem przekazu.

£¹czna suma

- **Przekazanych** — Ca³kowita suma przekazanych transakcji. Liczba ta nie obejmuje przekazów wymuszonych (przeprowadzonych rêcznie).
- **Przerwanych** — Ca³kowita suma przerwanych transakcji. Liczba ta nie obejmuje przerwañ wymuszonych.
- **Wymuszone przekazywanie** — Ca³kowita suma transakcji przekazanych rêcznie.
- **Wymuszone przerwanie** — Ca³kowita suma transakcji przerwanych rêcznie.
- **Razem** — Ca³kowita suma transakcji.

Czasy odpowiedzi

W tej grupie s¹ pokazane minimalne, œrednie i maksymalne czasy odpowiedzi transakcji w milisekundach. Czas odpowiedzi jest mierzony od momentu rozpoczêcia transakcji do momentu jej przekazania.

MS DTC uruchomiony/MS DTC zatrzymany

W tej grupie s¹ pokazane data i godzina rozpoczêcia bie¿¹cej sesji us³ugi MS DTC. Informacje te s¹ wyœwietlane dopiero po uruchomieniu us³ugi MS DTC. W grupie tej jest pokazywany równie¿ czas zatrzymania us³ugi MS DTC.

Niektóre statystyki maj¹ charakter zbiorczy; inne pokazuj¹ bie¿¹cy stan dzia³añ.

Po zatrzymaniu us³ugi MS DTC wartoœci wszystkich statystyk s¹ zerowane.

Zobacz te¿

Zarz¹dzanie us³ug¹ MS DTC

Komunikaty œledzenia

W oknie **Komunikaty œledzenia** jest wyœwietlana lista bie¿¹cych komunikatów œledzenia wysy³anych przez us³ugê Microsoft Distributed Transaction Coordinator (MS DTC). Mechanizm œledzenia pozwala sprawdzaæ bie¿¹cy stan ró¿nych dzia³añ us³ugi MS DTC (takich jak uruchamianie czy zakoñczenie pracy), a tak¿e ogl¹daæ dodatkowe informacje dotycz¹ce debugowania (przydatne przy wykrywaniu Ÿróde³ ró¿nych problemów).

Zagadnienia zwi¹zane z zarz¹dzaniem transakcjami opisano w nastêpuj¹cych tematach:

- Podstawowe informacje o transakcjach programu MTS
- Podstawowe informacje o stanach transakcji programu MTS

Korzystaj¹c z okna **Komunikaty œledzenia** mo¿na wykonywaæ nastêpuj¹ce zadania:

- Monitorowanie transakcji programu MTS
- Monitorowanie transakcji programu MTS w systemie Windows 95

Aby okreœliæ pokazywany w oknie poziom œledzenia, mo¿na przejœæ do arkusza w³aœciwoœci komputera i skorzystaæ z suwaka **Analiza** karty **Zaawansowane**.

Waga

Ikona	Opis
	B³êdy Wydarzy³o siê coœ, co wymaga ponownego uruchomienia us³ugi MS DTC. Na przyk³ad wykryto uszkodzony plik dziennika.
	Ostrze¿enia Istnieje mo¿liwoœæ wyst¹pienia nieprawid³owego dzia³ania.
	Informacje Pojawi³y siê informacje o rzadko wystêpuj¹cych zdarzeniach, na przyk³ad uruchomieniu lub zakoñczeniu pracy.
	Œledzenie Pojawi³y siê informacje dotycz¹ce debugowania o takich zdarzeniach, jak nowe po³¹czenia z klientem czy nabór mened¿erów zasobów.

□ród³o

W polu jest wyœwietlane Ÿród³o komunikatu œledzenia:

- **SVC** — □ród³em komunikatu œledzenia jest us³uga MS DTC.
- **LOG** — □ród³em komunikatu œledzenia jest dziennik us³ugi MS DTC.
- **CM** — □ród³em komunikatu œledzenia jest mened¿er po³¹czeñ sieciowych us³ugi MS DTC.

Zarówno œledz¹cy, jak i dziennik zdarzeñ systemu Windows NT do³¹cza do komunikatu informacje o jego Ÿródle.

Komunikat

W polu jest wyœwietlany komunikat.

Zobacz te¿

Zarz¹dzanie us³ug¹ MS DTC

Ikony transakcji

W oknie **Lista transakcji** s¹ wyœwietlane nastêpuj¹ce ikony:

Ikona **Opis**

Aktywna

Transakcja zosta³a uruchomiona.

Przerywana

Transakcja ma zostaæ przerwana. Us³uga MS DTC powiadamia wszystkich uczestników, ¿e transakcja musi zostaæ przerwana.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

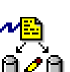
Przerwana

Transakcja zosta³a przerwana. Wszyscy uczestnicy zostali powiadomieni. Po przerwaniu transakcja jest natychmiast usuwana z listy transakcji wyœwietlanej w oknie **Transakcje MS DTC**.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

W przygotowaniu

Aplikacja klienta wys³a³a ¿¹danie przekazania transakcji. Us³uga MS DTC zbiera informacje o przygotowaniach od wszystkich uczestników.



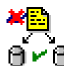
Przygotowana

Wszyscy uczestnicy odpowiedzieli twierdz¹co na pytanie o przygotowanie transakcji.

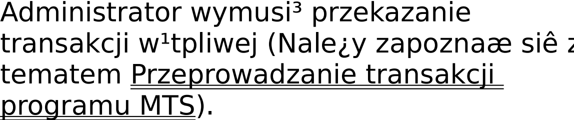
W¹tpliwa

Transakcja jest przygotowana, koordynuje j¹ inna us³uga MS DTC, a koordynuj¹ca us³uga MS DTC jest niedostêpna. Administrator systemu mo¿e wymusiæ przekazanie lub przerwanie transakcji, klikaj¹c w oknie **Transakcje** prawym przyciskiem myszy, a nastêpnie wybieraj¹c polecenia **Akcja/Przeka¿** lub **Akcja/Przerwij**. Po wymuszeniu wyniku transakcja jest traktowana jako "przekazana w sposób wymuszony" lub "przerwana w sposób wymuszony".

Uwaga Bez znajomoœci tematu Przeprowadzanie transakcj programu MTS nie nale¿y wymuszaæ rêcznie wyniku transakcji w¹tpliwej.



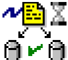
Wymuszone przekazanie



Administrator wymusi³ przekazanie transakcji w¹tpliwej (Nale¿y zapoznaæ siê z tematem Przeprowadzanie transakcji programu MTS).

Wymuszone przerwanie

Administrator wymusi³ przerwanie transakcji w¹tpliwej (Nale¿y zapoznaæ siê z tematem Przeprowadzanie transakcji).



Przekazywana

Transakcja zosta³a przygotowana pomyœlnie i us³uga MS DTC powiadamia uczestników, ¿e transakcja zosta³ przekazana. Us³uga MS DTC nie mo¿e zakoñczyæ transakcji, dopóki wszyscy uczestnicy nie potwierdz¹ odbioru (i zapisz¹ w dzienniku) ¿¹dania przekazania.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

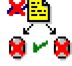
Nie mo¿na powiadomiæ o przerwaniu

Us³uga MS DTC powiadomi³a wszystkich pod³¹czonych uczestników, ¿e transakcja zosta³a przerwana. Nie zostali powiadomieni jedynie uczestnicy niedostêpni.

Ten stan transakcji wystêpuje wówczas, kiedy us³uga musi powiadomiæ mened¿era zasobów (na przyk³ad system IBM LU 6.2) o przerwaniu transakcji, ale nie mo¿e tego zrobiæ, poniewa¿ po³¹czenie z systemem firmy IBM zosta³o przerwane.

Administrator systemu mo¿e wymusiæ na us³udze MS DTC zapomnienie transakcji, klikaj¹c w oknie **Transakcje** prawym przyciskiem myszy, a nastêpnie wybieraj¹c polecenie **PrzeprowadŸ/Zapomnij**.

Uwaga Bez znajomoœci tematu Przeprowadzanie transakcji nie nale¿y wymuszaæ rêcznie zapominania transakcji.

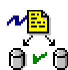


Nie mo¿na powiadomiæ o przekazaniu

Us³uga MS DTC powiadomi³a wszystkich pod³¹czonych uczestników, ¿e transakcja zosta³a przekazana. Nie zostali powiadomieni jedynie uczestnicy niedostêpni.

Administrator systemu mo¿e wymusiæ na us³udze MS DTC zapomnienie transakcji, klikaj¹c w oknie **Transakcje** prawym przyciskiem myszy, a nastêpnie wybieraj¹c polecenie **PrzeprowadŸ/Zapomnij**.

Uwaga Bez znajomoœci tematu Przeprowadzanie transakcji nie nale¿y wymuszaæ rêcznie zapominania transakcji.



Przekazana

Transakcja zosta³a przekazana, a wszyscy uczestnicy zostali o tym powiadomieni. Po przekazaniu transakcja jest natychmiast usuwana z listy transakcji wyœwietalnej w oknie **Transakcje MS DTC**.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

Zobacz te¿

Zarz¹dzanie us³ug¹ MS DTC

W³aœciwoœci komputera

W³aœciwoœci komputera okreœlaj¹ ogólne informacje o komputerze oraz wp³ywaj¹ na sposób aktualizacji komputera przez program Microsoft Transaction Server.

- **Ogólne**
- **Opcje**
- **Zaawansowane**

Zobacz te¿

Folder "Komputery"

Ogólne, karta (Komputer)

Na karcie **Ogólne** jest definiowana nazwa i opis komputera.

Nazwa

W tym polu jest wyœwietlana nazwa komputera.

Opis

W tym polu jest wyœwietlany opis komputera. Dziêki opisowi ³atwiej jest identyfikowaæ komputer i zarz¹dzaæ nim.

Zobacz te¿

Folder "Komputery"

Opcje, karta (Komputer)

Za pomoc¹ karty **Opcje** s¹ ustawiane w³aœciwoœci limitu czasu transakcji komputera oraz okreœlane informacje dla replikacji.

Limit czasu transakcji jest mierzony w sekundach i okreœla maksymalny czas aktywnoœci transakcji uruchamianych na tym komputerze. Transakcje, których aktywnoœæ trwa d³u¿ej ni¿ okreœlony limit czasu s¹ automatycznie przerywane przez system. Wartoœci¹ domyœln¹ limitu jest 60 sekund. Aby wy³¹czyæ limity czasu transakcji, nale¿y je wyzerowaæ; ustawienie to jest szczególnie u¿yteczne podczas debugowania aplikacji programu MTS.

Sekcja **Replikacja** karty **Opcje** s³u¿y do wprowadzania informacji o replikacji dla komputera z programem MTS. Aby zapewniæ obs³ugê bezawaryjn¹, w polu **Udzia³ replikacji** nale¿y wprowadziæ nazwê serwera wirtualnego, na którym dzia³a program Microsoft Cluster Server (MSCS). W przypadku komputerów wyposa¿onych w system Windows 95 replikacja katalogu programu MTS nie jest mo¿liwa.

Na karcie mo¿na okreœliæ równie¿ komputer, do którego bêd¹ mia³y zapewniony dostêp pliki wykonywalne klientów. W tym celu przed wygenerowaniem pliku wykonywalnego aplikacji w polu **Nazwa zdalnego serwera** nale¿y wprowadziæ nazwê fizycznego serwera, do którego bêd¹ mieli dostêp klienci. Jeœli pole to pozostanie puste, domyœlnie zostanie u¿yta nazwa fizycznego komputera eksportuj¹cego. Jeœli nazwa serwera zdalnego zostanie okreœlona jako ci¹g znakowy, wówczas wygenerowany przez program MTS Explorer plik wykonywalny aplikacji bêdzie wskazywa³ na tê w³aœnie nazwê serwera zdalnego.

Zobacz te¿

Korzystanie z replikacji programu MTS, Generowanie plików wykonywalnych programu MTS, Folder "Komputery"

Zaawansowane, karta (Komputer)

Karta **Zaawansowane** s³u¿y do konfigurowania w³aœciwoœci us³ugi Microsoft Distributed Transaction Coordinator (DTC). Wybrane ustawienia s¹ stosowane tylko do okien **Lista transakcji**, **Statystyki transakcji** i **Komunikaty œledzenia**.

Uwaga W ramach karty **Zaawaansowane** arkusza w³aœciwoœci **Mój komputer** nie mo¿na uzyskaæ dostêpu do dokumentacji Pomocy.

Widok

- **Odœwie¿anie wyœwietlania** — W³aœciwoœæ ta jest ustawiana za pomoc¹ suwaka, w zakresie od **Rzadko** do **Czêsto**. Ustawienie **Rzadko** oznacza, ¿e okna transakcji s¹ uaktualniane co 20 sekund, a ustawienie **Czêsto** oznacza, ¿e s¹ uaktualniane co sekunda. Wiêksza czêstotliwoœæ aktualizacji powoduje wiêksze obci¹¿enie transakcji zadaniami administracyjnymi, ale pozwala ogl¹daæ bardziej aktualne informacje.
- **Pokazywane transakcje** — W³aœciwoœæ ta jest ustawiana za pomoc¹ suwaka, w zakresie od **Bardzo stare** do **Nowe + Stare**. Wybór ustawienia **Bardzo stare** powoduje, ¿e wyœwietlane s¹ transakcje aktywne d³u¿ej ni¿ 5 minut, a wybór ustawienia **Nowe + Stare** powoduje wyœwietlanie transakcji aktywnych d³u¿ej ni¿ 1 sekunda.
- **Œledzenie** — W³aœciwoœæ ta jest ustawiana za pomoc¹ suwaka, w zakresie od **Mniej (przyspiesza MS DTC)** do **Wiêcej (zwalnia MS DTC)**. Mo¿na wyró¿niæ nastêpuj¹ce ustawienia:
  - Nie wysy³aj komunikatów œledzenia.
  - Wysy³aj tylko komunikaty œledzenia o b³êdach.
  - Wysy³aj komunikaty œledzenia o b³êdach i ostrzegawcze.
  - Wysy³aj komunikaty œledzenia o b³êdach, ostrzegawcze i informacyjne (ustawienie domyœlne).
  - Wysy³aj wszystkie komunikaty œledzenia.

W³aœciwoœci dziennika

- **Lokalizacja** — Miejsce przechowywania pliku dziennika.
- **Pojemnoœæ** — Maksymalne rozmiary pliku dziennika.

Zresetuj dziennik

Ustawienie powoduje aktualizacjê pliku dziennika po wprowadzeniu jakichkolwiek zmian.

Uwaga Nie nale¿y resetowaæ pliku dziennika us³ugi MS DTC, jeœli istniej¹ jakiekolwiek nie przeprowadzone transakcje.

Zagadnienia zwi¹zane z zarz¹dzaniem transakcjami opisano w nastêpuj¹cych tematach:

- Podstawowe informacje o transakcjach programu MTS
- Podstawowe informacje o stanach transakcji programu MTS

Wiêcej informacji na temat korzystania z okien programu MTS Explorer **Statystyki transakcji**, **Komunikaty œledzenia** i **Lista transakcji** mo¿na znaleŸæ pod has³ami:

- Monitorowanie transakcji programu MTS
- Monitorowanie transakcji programu MTS w systemie Windows 95

Zobacz te¿

Zarz¹dzanie us³ug¹ MS DTC

W³aœciwoœci pakietu

W³aœciwoœci pakietu okreœlaj¹ sposób dostêpu do pakietu.

- Ogólne
- Zabezpieczenia
- Zaawansowane
- To¿samoœæ
- Aktywacja

Wa¿ne W³aœciwoœci **Zabezpieczenia** i **To¿samoœæ** (lub sposób zamykania pakietu) nie mog¹ byæ modyfikowane za pomoc¹ arkuszy w³aœciwoœci dla pakietów bibliotek.

Jeœli komputer jest wyposa¿ony w system Windows 95, na arkuszach w³aœciwoœci pakietów nie wystêpuje karta **Zabezpieczenia**. Zabezpieczenia oparte na rolach s¹ obs³ugiwane tylko w systemie Windows NT.

Zobacz te¿

Folder "Zainstalowane pakiety"

Ogólne, karta (Pakiet)

Na karcie **Ogólne** s¹ wyœwietlane ogólne informacje o wybranym pakiecie.

Nazwa

W tym polu jest wyœwietlana nazwa pakietu.

Opis

W tym polu jest wyœwietlany opis pakietu. Dziêki opisowi ³atwiej jest identyfikowaæ pakiet i zarz¹dzaæ nim.

ID pakietu

W tym polu jest wyœwietlany numer identyfikacyjny pakietu, unikatowy numer generowany podczas tworzenia pakietu. Za pomoc¹ identyfikatora pakietu mo¿na identyfikowaæ na komputerze poszczególne wersje pakietu.

Zobacz te¿

Folder "Zainstalowane pakiety", W³aœciwoœci pakietu

Zabezpieczenia, karta (Pakiet)

Na karcie **Zabezpieczenia** s¹ wyœwietlane informacje o zabezpieczeniach wybranego pakietu.

W³¹cz sprawdzanie upowa¿nienia

Jeœli pole to jest zaznaczone, program Microsoft Transaction Server sprawdza poœwiadczenia zabezpieczeñ klientów wywo³uj¹cych pakiet. Sprawdzanie upowa¿nienia jest w³¹czane domyœlnie.

Poziom uwierzytelniania dla wywo³añ

Poziom uwierzytelniania dla klientów wywo³uj¹cych pakiet.

Zobacz te¿

Folder "Zainstalowane pakiety", W³aœciwoœci pakietu, Ustawianie w³aœciwoœci uwierzytelnieñ programu MTS, W³¹czanie zabezpieczeñ pakietu programu MTS

Zaawansowane, karta (Pakiet)

Karta **Zaawansowane** pozwala okreœliæ, czy skojarzony z pakietem proces serwera ma dzia³aæ zawsze czy te¿ byæ zamykany po up³ywie ustalonego czasu.

Aby pakiet by³ zamykany automatycznie, po ustalonym z góry czasie nieaktywnoœci, nale¿y wybraæ ustawienie **Zamknij, jeœli bezczynny od**.

Aby skojarzony z pakietem proces serwera by³ dostêpny zawsze, nale¿y wybraæ ustawienie **Pozostaw uruchomiony mimo bezczynnoœci**.

Aby zakoñczyæ wszystkie procesy serwera (uruchomione na wybranym komputerze), nale¿y u¿yæ polecenia **Zamknij proces serwera** z menu **Narzêdzia**.

Zobacz te¿

Folder "Zainstalowane pakiety"

To¿samoœæ, karta (Pakiet)

Karta **To¿samoœæ** s³u¿y do ustawiania to¿samoœci u¿ytkownika dla wszystkich sk³adników dzia³aj¹cych w danym pakiecie. Ustawieniem domyœlnym jest **U¿ytkownik interakcyjny**, który oznacza u¿ytkownika aktualnie zalogowanego na koncie serwera systemu Windows NT. Aby wybraæ innego u¿ytkownika, mo¿na zaznaczyæ opcjê **Ten u¿ytkownik**, a nastêpnie okreœliæ nazwê konta i has³o.

Wa¿ne Po wybraniu innego u¿ytkownika i okreœleniu has³a program Microsoft Transaction Server nie bêdzie sprawdzaæ poprawnoœci wprowadzanego has³a. Uruchomienie pakietu z niew³aœciwym has³em spowoduje b³¹d czasu wykonywania i przes³anie odpowiedniego komunikatu do dziennika zdarzeñ.

Aby ustawiæ opcjê **Ten u¿ytkownik** dla konkretnego u¿ytkownika lub grupê, trzeba zalogowaæ siê na komputerze, który albo jest zamapowany do tego u¿ytkownika, albo nale¿y do wybranej grupy.

Zobacz te¿

Konfiguracja ról programu MTS dla potrzeb zabezpieczeñ deklaratywnych, Folder "Zainstalowane pakiety"

Aktywacja, karta (Pakiet)

Karta **Aktywacja** s³u¿y do okreœlania sposobu uaktywniania sk³adników pakietu.

Pakiet mo¿e zostaæ uruchomiony albo w procesie wywo³uj¹cego go klienta (jako pakiet biblioteki), albo w wyspecjalizowanym procesie lokalnym (jako pakiet serwera).

Pakiet biblioteki

Opcjê tê nale¿y wybraæ, aby pakiet dzia³a³ jako pakiet biblioteki. Pakiet biblioteki jest uruchamiany w procesie klienta, który go utworzy³. Opcja ta jest dostêpna tylko dla klientów pracuj¹cych na komputerach, w których pakiet zainstalowano i skonfigurowano. Warto zauwa¿yæ, ¿e pakiety bibliotek nie daj¹ mo¿liwoœci korzystania ze œledzenia sk³adników, sprawdzania ról i mechanizmu izolowania procesów.

Pakiet serwera

Opcjê tê nale¿y wybraæ, aby pakiet dzia³a³ jako pakiet serwera. Pakiet serwera jest uruchamiany na komputerze lokalnym, w swoim w³asnym procesie. Pakiety serwera obs³uguj¹ zabezpieczenia oparte na rolach, wspó³u¿ytkowanie zasobów, izolowanie procesów i zarz¹dzanie procesami (na przyk³ad œledzenie pakietu).

Zobacz te¿

Folder "Zainstalowane pakiety", W³aœciwoœci pakietu

W³aœciwoœci sk³adnika

W³aœciwoœci sk³adnika obejmuj¹ sposób obs³ugi transakcji oraz ustawienia zabezpieczeñ. Za pomoc¹ karty **W³aœciwoœci** mo¿na ponadto przegl¹daæ w³aœciwoœci identyfikuj¹ce sk³adnik, na przyk³ad jego nazwê, identyfikator programistyczny (ProgID) oraz identyfikator klasy (CLSID).

- **Ogólne**
- **Transakcja**
- **Zabezpieczenia**

Zobacz te¿

Folder "Sk³adniki", W³¹czanie zabezpieczeñ pakietu programu MTS

Ogólne, karta (Sk³adnik)

Na karcie **Ogólne** s¹ wyœwietlane ogólne informacje o wybranym sk³adniku.

ProgID sk³adnika

W tym polu jest wyœwietlany identyfikator programistyczny (ProgID).

Opis

W tym polu jest wyœwietlany opis sk³adnika. Dziêki opisowi ³atwiej jest identyfikowaæ sk³adnik i zarz¹dzaæ nim.

DLL

W tym polu jest wyœwietlana œcie¿ka dostêpu do biblioteki DLL zawieraj¹cej definicje klasy i interfejsu sk³adnika.

CLSID

W tym polu jest wyœwietlany unikatowy identyfikator klasy (CLSID) wybranego sk³adnika. Identyfikatora CLSID mo¿na u¿ywaæ w kodzie programów w celu identyfikowania sk³adnika i uzyskiwania do niego dostêpu.

Pakiet

W tym polu jest wyœwietlana nazwa pakietu, w którym zainstalowano wybrany sk³adnik.

Zobacz te¿

Folder "Sk³adniki", W³aœciwoœci sk³adnika

Zabezpieczenia, karta (Sk³adnik)

Karta **Zabezpieczenia** s³u¿y do konfigurowania zabezpieczeñ wybranego sk³adnika. Wiêcej informacji na temat zabezpieczeñ deklaratywnych mo¿na znaleŸæ pod has³em W³¹czanie zabezpieczeñ pakietu programu MTS.

W³¹cz sprawdzanie upowa¿nienia

Po zaznaczeniu tego pola bêd¹ sprawdzane uwierzytelnienia wszystkich klientów wywo³uj¹cych sk³adnik.

Zobacz te¿

Folder "Sk³adniki", W³aœciwoœci sk³adników programu MTS

Transakcja, karta (Sk³adnik)

Karta **Transakcja** pozwala okreœliæ sposób obs³ugi transakcji przez sk³adnik

- **Wymaga transakcji** — Obiekty sk³adnika musz¹ zostaæ wykonane w zakresie transakcji. Kiedy jest tworzony nowy obiekt, jego kontekst dziedziczy transakcjê od kontekstu klienta. Jeœli klient nie ma transakcji, program MTS automatycznie tworzy dla obiektu now¹ transakcjê.
- **Wymaga nowej transakcji** — Obiekty sk³adnika musz¹ byæ wykonywane w ramach swoich w³asnych transakcji. Kiedy jest tworzony nowy obiekt, program MTS *automatycznie* tworzy dla niego now¹ transakcjê, niezale¿nie od tego, czy klient ma transakcjê.
- **Obs³uguje transakcje** — Obiekty sk³adnika mog¹ byæ wykonywane w ramach transakcji klientów. Kiedy jest tworzony nowy obiekt, jego kontekst dziedziczy transakcjê od kontekstu klienta. Jeœli klient nie ma transakcji, nowy kontekst jest tworzony równie¿ bez transakcji.
- **Nie obs³uguje transakcji** — Obiekty sk³adnika nie powinny byæ wykonywane w zakresie transakcji. Kiedy powstaje nowy obiekt, jego kontekst jest tworzony bez transakcji, niezale¿nie od tego, czy klient ma transakcjê.

Zobacz te¿

Folder "Sk³adniki", W³aœciwoœci sk³adników programu MTS, Zarz¹dzanie transakcjami programu MTS

W³aœciwoœci sk³adnika zdalnego

W³aœciwoœci sk³adnika zdalnego s³u¿¹ do wyœwietlania informacji o sk³adnikach dodanych do folderu "Sk³adniki zdalne". Na karcie **Ogólne** nie mo¿na konfigurowaæ innych w³aœciwoœci sk³adników zdalnych poza ich opisem.

Ogólne, karta

Zobacz te¿

Folder "Sk³adniki zdalne", Rozpowszechnianie pakietów programu MTS

Ogólne, karta (Sk³adnik zdalny)

Na karcie **Ogólne** s¹ wyœwietlane informacje identyfikuj¹ce wybrany sk³adnik zdalny.

Nazwa

W tym polu jest wyœwietlana nazwa sk³adnika zdalnego.

Opis

W tym polu jest wyœwietlany opis sk³adnika zdalnego. Dziêki opisowi ³atwiej jest identyfikowaæ sk³adnik zdalny i zarz¹dzaæ nim.

CLSID

W tym polu jest wyœwietlany identyfikator klasy sk³adnika.

Uruchamiany na

W tym polu jest wyœwietlana nazwa komputera, z którego zainstalowano sk³adnik.

Zobacz te¿

Folder "Sk³adniki zdalne"

W³aœciwoœci roli

W³aœciwoœci roli pozwalaj¹ obejrzeæ nazwê, opis i identyfikator roli.

Ogólne

Zobacz te¿

Folder "Role", W³¹czanie zabezpieczeñ pakietu programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup

Ogólne, karta (Rola)

Na karcie **Ogólne** s¹ wyœwietlane ogólne informacje o wybranej roli.

Nazwa

W tym polu jest wyœwietlana nazwa roli.

Opis

W tym polu jest wyœwietlany opis roli. Dziêki opisowi ³atwiej jest identyfikowaæ rolê i zarz¹dzaæ ni¹.

ID roli

W tym polu jest wyœwietlany numer identyfikacyjny roli, generowany przez program MTS podczas dodawania roli. Za pomoc¹ tego identyfikatora mo¿na odró¿niæ role o tych samych nazwach, odnosz¹ce siê jednak do ró¿nych pakietów.

Informacje na temat obiektów z folderu "Role" oraz funkcji wykonywanych na poziomie ról mo¿na znaleŸæ pod has³em Folder "Role".

Zobacz te¿

Folder "Role", W³aœciwoœci ról programu MTS, W³¹czanie zabezpieczeñ pakietu programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup

W³aœciwoœci elementu roli

W³aœciwoœci elementu roli s³u¿¹ do wyœwietlania informacji o rolach, które dodano do sk³adnika lub interfejsu.

Ogólne

Zobacz te¿

Folder "Przynale¿noœæ ról", W³¹czanie zabezpieczeñ pakietu programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup

Ogólne, karta (Element roli)

Na karcie **Ogólne** s¹ wyœwietlane ogólne informacje o wybranej roli.

Nazwa

W tym polu jest wyœwietlana nazwa roli.

Opis

W tym polu jest wyœwietlany opis roli. Dziêki opisowi ³atwiej jest identyfikowaæ rolê i zarz¹dzaæ ni¹.

ID roli

W tym polu jest wyœwietlany numer identyfikacyjny roli, generowany przez program MTS podczas dodawania roli. Za pomoc¹ tego identyfikatora mo¿na odró¿niæ role o tych samych nazwach, odnosz¹ce siê jednak do ró¿nych pakietów.

Zobacz te¿

Folder "Przynale¿noœæ ról", W³aœciwoœci roli, W³¹czanie zabezpieczeñ pakietu programu MTS, Mapowanie ról programu MTS do u¿ytkowników i grup

W³aœciwoœci interfejsu

W³aœciwoœci interfejsu s³u¿¹ do wyœwietlania informacji o interfejsie wykorzystywanym przez sk³adnik.

- **Ogólne**
- **Proxy**

Zobacz te¿

Folder "Interfejsy"

Ogólne, karta (Interfejs)

Na karcie **Ogólne** s¹ wyœwietlane informacje identyfikuj¹ce wybrany interfejs.

Nazwa

W tym polu jest wyœwietlana nazwa interfejsu.

Opis

W tym polu jest wyœwietlany opis interfejsu. Dziêki opisowi ³atwiej jest identyfikowaæ interfejs i zarz¹dzaæ nim.

IID

W tym polu jest wyœwietlany identyfikator interfejsu.

Zobacz te¿

Folder "Interfejsy", W³aœciwoœci interfejsu

Proxy, karta (Interfejs)

Na karcie s¹ wyœwietlane informacje identyfikuj¹ce wybrany interfejs typu proxy/stub.

Proxy/Stub

W tym polu jest wyœwietlany identyfikator CLSID oraz nazwa pliku biblioteki DLL typu proxy/stub.

Biblioteka typów

W tym polu jest wyœwietlany identyfikator biblioteki oraz lokalizacja biblioteki typów.

Zobacz te¿

Folder "Interfejsy", W³aœciwoœci interfejsu

W³aœciwoœci metody

W³aœciwoœci metody s³u¿¹ do wyœwietlania informacji o metodach wykorzystywanych przez interfejs.

Ogólne

Zobacz te¿

Folder "Metody"

Ogólne, karta (Metoda)

Na karcie **Ogólne** s¹ wyœwietlane informacje identyfikuj¹ce wybran¹ metodê.

Nazwa

W tym polu jest wyœwietlana nazwa metody.

Opis

W tym polu mo¿na wpisaæ opis metody.

Zobacz te¿

Folder "Metody", W³aœciwoœci metody

Tworzenie pakietów MTS

Tworzenie pakietów jest ostatnim elementem procesu projektowania aplikacji MTS. Decyzje podejmowane przy projektowaniu pakietu okreœlaj¹ sposób organizacji i w³aœciwoœci sk³adników. Mimo ¿e tworzenie pakietów MTS nie wymaga wiedzy programistycznej, ich twórcy powinni byæ zaznajomieni ze specyfikacjami u¿ywanymi przy projektowaniu i implementacji aplikacji.

Niniejszy podrozdzia³ *Podrêcznika administratora programu MTS* powinien byæ czytany w po³¹czeniu z odpowiednimi rozdzia³ami *Podrêcznika programisty progarmu MTS*. Dziêki temu u¿ytkownik zyska pe³niejsz¹ wiedzê na temat konsekwencji (projektowych) grupowania sk³adników w pakiety za pomoc¹ programu Microsoft Transaction Server Explorer.

Grupuj¹c sk³adniki w pakiety, projektant musi podj¹æ szereg decyzji dotycz¹cych podzia³u zasobów na pule, ustawieñ aktywacji oraz sposobu obs³ugi transakcji. Na przyk³ad podczas tworzenia pakietu sk³adniki powinno siê pogrupowaæ w ten sposób, aby wspó³u¿ytkowa³y zasoby z tego samego pakietu. Ponadto powinno siê ustaliæ typy zasobów wspó³u¿ytkowanych przez sk³adniki i pogrupowaæ sk³adniki korzystaj¹ce z "drogich" zasobów, na przyk³ad po³¹czeñ z konkretn¹ baz¹ danych.

Twórz¹c pakiety w sposób zapewniaj¹cy odpowiedni rozdzia³ zasobów, mo¿na projektowaæ bardziej wydajne aplikacje MTS. Po³¹czenie informacji z *Podrêcznika programisty programu MTS* i *Podrêcznika administratora programu MTS* pozwoli lepiej zrozumieæ zalety tworzenia i grupowania pakietów za pomoc¹ programu MTS Explorer. Informacje na temat sposobów uzyskania *Podrêcznika administratora programu MTS* mo¿na znaleŸæ pod has³em Instalacja dokumentacji i przyk³ady projektów MTS.

W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Tworzenie pustego pakietu MTS

Dodawanie sk³adnika do pakietu MTS

Import sk³adnika do pakietu MTS

Usuwanie sk³adnika z pakietu MTS

Budowanie pakietu MTS w celu eksportu

Ustawianie w³aœciwoœci pakietu MTS

Ustawianie w³aœciwoœci aktywacji MTS

Ustawianie w³aœciwoœci transakcji MTS

Ustawianie poziomów uwierzytelnienia MTS

Blokowanie pakietu MTS

Tworzenie pustego pakietu MTS

Pierwszym krokiem, jaki nale¿y wykonaæ w programie MTS Explorer jest utworzenie pakietu. Pakiet (który za pomoc¹ programu MTS tworzy siê bardzo ³atwo) jest kolekcj¹ sk³adników dzia³aj¹cych w tym samym procesie. U¿ytkownik programu mo¿e albo utworzyæ pusty pakiet i dopiero potem dodaæ do niego sk³adniki, albo zainstalowaæ pakiet wstêpnie wbudowany. Pakiety mo¿na dodawaæ za pomoc¹ Kreatora pakietów lub przeci¹gaj¹c plik pakietu (.pak) z programu Eksplorator Windows NT i upuszczaj¹c go w prawym okienku programu MTS Explorer.

Pakiety definiuj¹ granice dla procesu serwera dzia³aj¹cego na komputerze serwera. Na przyk³ad jeœli sk³adnik sprzedawcy i sk³adnik nabywcy zostan¹ umieszczone w dwóch ró¿nych pakietach, obydwa sk³adniki bêd¹ uruchomione w oddzielnych procesach izolowanych. Z tego powodu, jeœli jeden z procesów serwera zostanie niespodziewanie przerwany (na przyk³ad w wyniku krytycznego b³êdu aplikacji), drugi pakiet bêdzie realizowany nadal w swoim oddzielnym procesie.

Aby utworzyæ pusty pakiet

1. W lewym okienku programu MTS Explorer zaznacz komputer, dla którego chcesz utworzyæ pakiet.
2. W ramach tego komputera otwórz folder "Zainstalowane pakiety".
3. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie¿ zaznaczyæ folder "Zainstalowane pakiety", a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ z menu kolejno polecenia **Nowy** i **Pakiet**, albo na pasku narzêdzi MTS klikn¹æ przycisk **Utwórz nowy obiekt**.
4. Za pomoc¹ Kreatora pakietów zainstaluj pakiet wstêpnie wbudowany lub utwórz pusty pakiet. Aby utworzony pusty pakiet sta³ siê funkcjonalny, musisz dodaæ doñ sk³adniki i role.
5. Kliknij przycisk **Utwórz pusty pakiet**.
6. Wpisz nazwê nowego pakietu i kliknij przycisk **Dalej**.
7. W oknie dialogowym **Ustaw to¿samoœæ pakietu** ustaw to¿samoœæ pakietu, a nastêpnie kliknij przycisk **Zakoñcz**.

Ustawieniem domyœlnym to¿samoœci pakietu jest **U¿ytkownik interakcyjny**. U¿ytkownik interaktywny oznacza u¿ytkownika zalogowanego na tym komputerze serwera, na którym jest uruchomiony pakiet. Aby okreœliæ innego rodzaju u¿ytkownika, mo¿na skorzystaæ z opcji **Ten u¿ytkownik** i podaæ konkretnego u¿ytkownika lub grupê systemu Windows NT.

Zobacz te¿

Dodawanie sk³adnika do pakietu MTS, Import sk³adnika do pakietu MTS, Budowanie pakietu MTS w celu eksportu, W³¹czanie zabezpieczenia pakietu MTS

Dodawanie sk³adnika do pakietu MTS

Sk³adnik MTS jest to fragment kodu i danych wielokrotnego u¿ytku wbudowany w specyfikacjê modelu COM (z ang. Component Object Model). Sk³adniki wnosz¹ do aplikacji logikê pracy.

Sk³adnik mo¿na dodaæ do pakietu:

- korzystaj¹c z kreatora sk³adników.
- przenosz¹c sk³adnik z istniej¹cego pakietu.

Aby dodaæ sk³adniki do folderu "Sk³adniki" pakietu mo¿na: albo u¿yæ kreatora sk³adników programu MTS Explorer, albo przeci¹gn¹æ bibliotekê do³¹czan¹ dynamicznie (DLL) zawieraj¹c¹ po¿¹dane sk³adniki z okna Eksploratora Windows NT i upuœciæ j¹ w pakiecie. Jeœli wykorzystywany jest Kreator sk³adników mo¿na: albo zainstalowaæ nowy sk³adnik (co spowoduje dodanie sk³adnika do rejestru systemowego), albo zaimportowaæ sk³adniki ju¿ zarejestrowane. Sk³adniki mog¹ byæ dodawane zarówno do pustych, jak i istniej¹cych pakietów.

Aby przenieœæ sk³adnik z istniej¹cego pakietu, wystarczy po prostu przeci¹gn¹æ go z istniej¹cego pakietu do nowego sk³adnika i upuœciæ.

Warto zauwa¿yæ, ¿e pojedyncza aplikacja MTS mo¿e zawieraæ sk³adniki, które mo¿na instalowaæ w wielu pakietach. Ró¿ne sk³adniki znajduj¹ce siê w tej samej bibliotece DLL mo¿na umieszczaæ w zupe³nie ró¿nych pakietach.

Aby dodaæ sk³adnik do pakietu

1. W lewym okienku programu MTS Explorer zaznacz komputer, dla którego chcesz utworzyæ pakiet.
2. Otwórz folder "Zainstalowane pakiety" i zaznacz pakiet, w którym zamierzasz zainstalowaæ sk³adnik.
3. Otwórz folder "Sk³adniki".
4. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie¿ zaznaczyæ folder "Zainstalowane sk³adniki", a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ z menu kolejno polecenia **Nowy** i **Sk³adnik**, albo na pasku narzêdzi MTS klikn¹æ przycisk **Utwórz nowy obiekt**.
5. Kliknij przycisk **Instaluj nowe sk³adniki**.
6. W wyœwietlonym oknie dialogowym kliknij przycisk **Dodaj pliki,** aby zaznaczyæ pliki wybrane do instalacji. Nale¿y zaznaczyæ bibliotekê DLL zawieraj¹c¹ instalowany sk³adnik. Jeœli sk³adnik zawiera zewnêtrzn¹ bibliotekê typów lub bibliotekê DLL typu proxy/stub, dodaj równie¿ te pliki. SprawdŸ, czy w programie Eksplorator systemu Windows NT opcjê **Pliki ukryte** ustawiono jako **Poka¿ wszystkie pliki**. Jeœli opcja ta jest ustawiona tak, aby ukrywane by³y pliki z rozszerzeniem .dll, wówczas w oknie dialogowym **Dodaj pliki** Kreatora sk³adników nie bêd¹ widoczne biblioteki DLL zawieraj¹ce sk³adnik. Po zmianie opcji nale¿y ponownie uruchomiæ program MTS Explorer.
7. W wyœwietlonym oknie dialogowym zaznacz plik lub pliki, które zamierzasz dodaæ, a nastêpnie kliknij przycisk **Otwórz**. Ustawiaj¹c odpowiedni¹ opcjê w polu **Pliki typu** mo¿esz wyœwietlaæ wszystkie dostêpne pliki, tylko biblioteki DLL lub tylko biblioteki typów.
8. Po dodaniu plików w oknie dialogowym **Instaluj sk³adniki** zostan¹ wyœwietlone dodane pliki oraz skojarzone z nimi sk³adniki. Jeœli zaznaczysz pole wyboru **Szczegó³y**, zostan¹ wyœwietlone bardziej szczegó³owe informacje o zawartoœci plików i o znalezionych sk³adnikach. Sk³adniki programu Microsoft Transaction Server musz¹ zawieraæ bibliotekê typów. Jeœli program nie bêdzie w stanie odnaleŸæ

biblioteki typów, dany sk³adnik nie zostanie pokazany na liœcie. Aby usun¹æ plik z listy **Pliki do zainstalowania**, mo¿esz go zaznaczyæ i klikn¹æ przycisk **Usuñ pliki**.

9 Kliknij przycisk **Zakoñcz**, aby zainstalowaæ sk³adnik. Tutaj warto wspomnieæ, ¿e instalacja sk³adnika pozwala ogl¹daæ jego interfejsy i metody. Jeœli sk³adnik jest importowany, jego interfejsy i metody nie s¹ widoczne w programie MTS Explorer.

Zobacz te¿

Tworzenie pustego pakietu MTS, Import sk³adnika do pakietu MTS, Usuwanie sk³adnika z pakietu MTS, Budowanie pakietu MTS w celu eksportowania

Import sk³adnika do pakietu MTS

Program Microsoft Transaction Server Explorer pozwala importowaæ do pakietów konkretne sk³adniki, które uprzednio zosta³y zarejestrowane na komputerze jako sk³adniki COM (z ang. Component Object Model). Import sk³adnika nie powoduje zainstalowania informacji o interfejsie lub metodach, niezbêdnych do ustawienia w³aœciwoœci interfejsu lub skonfigurowania sposobu dostêpu do sk³adnika ze strony klientów zdalnych. Jeœli to mo¿liwe, sk³adniki nale¿y raczej instalowaæ ni¿ importowaæ.

Aby zaimportowaæ sk³adnik do pakietu

1. W lewym okienku programu MTS Explorer zaznacz komputer, do którego chcesz zaimportowaæ sk³adnik.
2. Otwórz folder "Zainstalowane pakiety" i zaznacz pakiet, do którego chcesz importowaæ sk³adnik.
3. Otwórz folder "Sk³adniki".
4. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie¿ zaznaczyæ folder "Sk³adniki", a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ z menu kolejno polecenia **Nowy** i **Sk³adnik**, albo na pasku narzêdzi MTS klikn¹æ przycisk **Utwórz nowy obiekt**.
5. Kliknij przycisk **Importowaæ sk³adniki, które s¹ ju¿ zarejestrowane**.
6. Zaznacz sk³adniki, które chcesz importowaæ.
7. Kliknij przycisk **Zakoñcz**.

Zobacz te¿

Tworzenie pustego pakietu MTS, Dodawanie sk³adnika do pakietu MTS

Usuwanie sk³adnika MTS z pakietu

Program Microsoft Transaction Server Explorer pozwala usuwaæ sk³adniki z pakietu. Konsekwencje usuniêcia sk³adnika zale¿¹ od sposobu dodawania sk³adnika do œrodowiska czasu wykonania programu MTS. Jeœli sk³adnik by³ instalowany, jego usuniêcie spowoduje ca³kowite wykasowanie informacji rejestru zarówno ze œrodowiska czasu wykonywania programu MTS, jak i z pamiêci komputera. Jeœli sk³adnik by³ importowany, zostanie on wprawdzie usuniêty ze œrodowiska czasu wykonywania programu MTS, ale pozostanie zarejestrowany na komputerze rozmieszczaj¹cym jako sk³adnik COM (z ang. Component Object Model). Nale¿y wówczas rêcznie usun¹æ z pamiêci komputera wpisy do rejestru sk³adnika COM oraz pliki sk³adnika.

Aby usun¹æ sk³adnik z pakietu

1. W prawym okienku programu MTS Explorer zaznacz pakiet zawieraj¹cy sk³adnik, który chcesz usun¹æ.
2. Otwórz folder "Sk³adniki".
3. Zaznacz sk³adnik, który chcesz usun¹æ.
4. W menu **Akcja** kliknij polecenie **Usuñ**. Mo¿esz równie¿ zaznaczyæ sk³adnik, a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ z menu polecenie **Usuñ**, albo na pasku narzêdzi MTS klikn¹æ przycisk **Usuñ**.
5. W wyœwietlonym oknie dialogowym kliknij przycisk **Tak**.

Zobacz te¿

Tworzenie pustego pakietu MTS

Budowanie pakietu MTS w celu eksportowania

Eksportowanie pakietów wymaga poprawnego skonfigurowania sk³adników tak, aby rejestracja pakietu na komputerze serwera lub komputerze klienta mog³a przebiegaæ automatycznie. Z tego powodu podczas budowania pakietu, nale¿y uwzglêdniæ mo¿liwe sposoby przysz³ego rozpowszechniania pakietu.

Aby wyeksportowaæ pakiet do innego komputera serwera, na którym dzia³a program MTS, mo¿na skorzystaæ z opcji eksportu pakietów programu Microsoft Transaction Server Explorer. Mo¿na równie¿ generowaæ pliki wykonywalne dla komputerów klientów zdalnych, na których dzia³a system Windows NT lub Windows 95 (z obs³ug¹ modelu DCOM), umo¿liwiaj¹ce dostêp do aplikacji serwera. Podczas tworzenia i konfigurowania pakietów nale¿y uwzglêdniæ wymagania zwi¹zane z ich eksportem.

Domyœlnie pliki wykonywalne powoduj¹ takie skonfigurowanie komputerów klientów, aby by³ zapewniony dostêp do zdalnego serwera MTS, na którym wygenerowano plik wykonywalny. Lokalizacjê aplikacji serwera mo¿na modyfikowaæ, konfiguruj¹c kartê **Opcja** arkuszy w³aœciwoœci komputera. Przed wygenerowaniem pliku wykonywalnego nale¿y zaznaczyæ w programie MTS Explorer ikonê **Mój Komputer,** klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **W³aœciwoœci**. Nastêpnie nale¿y klikn¹æ kartê **Opcje**, po czym w polu **Nazwa serwera zdalnego** wprowadziæ nazwê serwera, do którego bêd¹ mia³y dostêp komputery kientów. Warto zauwa¿yæ, ¿e podana nazwa serwera musi siê odnosiæ do serwera MTS, na którym dzia³a pakiet. Na sam koniec nale¿y klikn¹æ przycisk **OK**. Po wygenerowaniu pliku wykonywalnego dla klienta plik ten tak skonfiguruje komputer klienta, aby mia³ on dostêp do serwera o nazwie okreœlonej w polu **Nazwa serwera zdalnego**. Dziêki temu mo¿na bêdzie tak opracowaæ aplikacjê, aby wielu klientów mog³o odwo³ywaæ siê do kilku komputerów, na których jest uruchomiony ten sam pakiet.

Wymagania eksportu pakietów

Projektanci pakietów oraz administratorzy zaawansowanych systemów i sieci Web, buduj¹c i rozmieszczaj¹ cpakiety MTS, powinni mieæ na wzglêdzie nastêpuj¹ce wymagania:

- Z bibliotek typów aplikacji czysto klienckich nale¿y usuwaæ opisy standardowych interfejsów COM. Na przyk³ad projektant pakietu móg³ zdefiniowaæ w bibliotece typów interfejs **IObjectSaftey** z myœl¹ o przysz³ym korzystaniu zeñ w ramach programu Visual Basic. Usuniêcie opisów interfejsu przed rozpoczêciem eksportu zapobiegnie nieprawid³owemu rejestrowaniu i odrejestrowywaniu pakietu na komputerach klientów. Nie usuniêcie opisów standardowych interfejsów COM z czysto klienckiej biblioteki typów spowoduje awariê ka¿dej innej aplikacji korzystaj¹cej z tych interfejsów.
- Jeœli którykolwiek z unikatowych identyfikatorów globalnych (identyfikatorów GUID) zawartych w pakiecie serwera i u¿ywanych przez klientów (miêdzy innymi identyfikatorów bibliotek typów, interfejsów i klas) ulegnie zmianie, wówczas aby wygenerowaæ dla klienta uaktualniony instalacyjny plik wykonywalny, nale¿y ponownie wyeksportowaæ pakiet. Klienci aplikacji nie bêd¹ w stanie uzyskaæ dostêpu do aplikacji serwera, dopóki nie uruchomi¹ nowego instalacyjnego pliku wykonywalnego. Warto zauwa¿yæ, ¿e niektóre narzêdzia projektowe (na przyk³ad program Microsoft® Visual Basic™) mog¹ zmieniaæ identyfikatory GUID bez powiadamiania projektanta.

Aby uzyskaæ wiêcej informacji o stosowaniu programu MTS Explorer do rozpowszechniania pakietów MTS, zapoznaj siê z treœci¹ podrozdzia³u Rozpowszechnianie pakietów MTS.

Zobacz te¿

Tworzenie pustego pakietu MTS, Dodawanie sk³adnika do pakietu MTS, Importowanie sk³adnika do pakietu MTS

Ustawianie w³aœciwoœci pakietu MTS

W³aœciwoœci pakietu okreœlaj¹ konfiguracjê pakietu. Aby uzyskaæ dostêp do arkuszy w³aœciwoœci, mo¿na w programie Microsoft Transaction Server Explorer klikn¹æ pakiet prawym przyciskiem myszy lub zaznaczyæ pakiet i w menu **Akcja** wybraæ opcjê **W³aœciwoœci**.

Mo¿na wyró¿niæ piêæ arkuszy w³aœciwoœci pakietu:

- **Ogólne**

  Na arkuszu jest wyœwietlana nazwa i opis komputera.

- **Zabezpieczenia**

  Pozwala w³¹czyæ sprawdzanie upowa¿nienia. Ustawienie domyœlne powoduje w³¹czenie sprawdzania upowa¿nienia. Wiêcej informacji na ten temat mo¿na uzyskaæ pod has³em W³¹czanie zabezpieczenia pakietu MTS.

- **Zaawansowane**

  Pozwala okreœliæ, czy skojarzony z pakietem proces serwera ma dzia³aæ zawsze, czy zamkn¹æ go po up³ywie okreœlonego czasu.

- **To¿samoœæ**

  Pozwala okreœliæ u¿ytkowników, którzy mog¹ uzyskiwaæ dostêp do pakietu. Wartoœci¹ domyœln¹ jest **U¿ytkownik interakcyjny**, co oznacza u¿ytkownika aktualnie zalogowanego na koncie serwera systemu Windows NT. Aby wybraæ innego u¿ytkownika, nale¿y zaznaczyæ opcjê **Ten u¿ytkownik** oraz podaæ nazwê konta i has³o.

- **Aktywacja**

  Pozwala okreœliæ poziom aktywacji dla pakietu i jego sk³adników. Poziom aktywacji mo¿na ustawiæ jako **Pakiet biblioteki**, co powoduje uaktywnianie sk³adników w procesie kreatora, lub jako **Pakiet serwera**, co powoduje uruchomienie pakietu w procesie specjalizowanego serwera.

  Wiêcej informacji na ten temat mo¿na znaleŸæ pod has³em Ustawianie w³aœciwoœci aktywacji MTS.

Do modyfikacji w³aœciwoœci pakietu lub sk³adnika s³u¿¹ arkusze w³aœciwoœci.

Aby uzyskaæ dostêp do arkuszy w³aœciwoœci:

1. Kliknij prawym przyciskiem myszy pakiet lub sk³adnik, który zamierzasz skonfigurowaæ, a nastêpnie kliknij polecenie **W³aœciwoœci**. W tym celu mo¿esz równie¿ zaznaczyæ element, otworzyæ menu **Akcja** i wybraæ polecenie **W³aœciwoœci**.
2. Zaznacz kartê arkusza, której zamierzasz u¿yæ.
3. Uaktualnij ustawienie w³aœciwoœci.
4. Kliknij przycisk **OK,** aby zapisaæ ustawienie i powróciæ do programu MTS Explorer.

Odœwie¿anie ustawieñ sk³adnika

Bardzo wa¿n¹ czynnoœci¹ jest odœwie¿anie ustawieñ MTS sk³adników po ka¿dej kolejnej kompilacji projektu. Odœwie¿anie ustawieñ sk³adnika zapobiega ponownemu zapisywaniu ustawieñ rejestru sk³adnika.

Aby odœwie¿yæ ustawienia sk³adnika

1. W lewym okienku programu MTS Explorer zaznacz komputer zawieraj¹cy sk³adniki przeznaczone do odœwie¿enia.
2. W menu **Akcja** kliknij polecenie **Odœwie¿ wszystkie sk³adniki**. Polecenie powoduje uaktualnienie programu Transaction Server ze wzglêdu na wszystkie zmiany wprowadzone w rejestrze systemu, identyfikatorach CLSID sk³adnika oraz

identyfikatorach interfejsów (IID). Sk³adniki mo¿na odœwie¿yæ równie¿, zaznaczaj¹c komputer w lewym okienku programu MTS Explorer, po czym klikaj¹c przycisk **Odœwie¿** z paska narzêdzi MTS.

Polecenie **Odœwie¿ wszystkie sk³adniki** jest wykonywane, jeœli w programie MTS Explorer zaznaczono jakikolwiek element w drzewie pod komputerem. Polecenie stosuje siê do zaznaczonego komputera.

Zobacz te¿

Ustawianie w³aœciwoœci aktywacji MTS, Ustawianie w³aœciwoœci transakcji MTS, Ustawianie poziomów uwierzytelnienia MTS, Blokowanie pakietu MTS

Ustawianie w³aœciwoœci aktywacji MTS

W³aœciwoœci aktywacji powinny byæ okreœlane w czasie projektowania pakietu. Pakiet mo¿na ustawiæ tak, aby dzia³a³ w procesie wywo³uj¹cego go klienta (jako pakiet biblioteki) lub w procesie specjalizowanego serwera (jako pakiet serwera).

Pakiet biblioteki

Po wyborze tej opcji pakiet jest uruchamiany jako pakiet biblioteki. Pakiet taki dzia³a w procesie klienta, który go utworzy³. Opcja jest dostêpna tylko dla klientów korzystaj¹cych z komputera, na którym pakiet zosta³ zainstalowany i skonfigurowany. Pakiety bibliotek nie daj¹ mo¿liwoœci œledzenia pracy sk³adników, sprawdzania ról oraz izolowania procesów.

Pakiet serwera

Po wyborze tej opcji pakiet jest uruchamiany jako pakiet serwera. Pakiet taki dzia³a w swoim w³asnym procesie na komputerze lokalnym. Pakiety serwerów daj¹ mo¿liwoœæ korzystania z zabezpieczeñ opartych na rolach, wspó³u¿ytkowania zasobów, izolowania procesów oraz zarz¹dzania procesami (na przyk³ad œledzenia pakietów).

Aby ustawiæ w³aœciwoœci aktywacji pakietu

1. Zaznacz pakiet, który chcesz skonfigurowaæ.
2. W menu **Akcja** kliknij polecenie **W³aœciwoœci** i zaznacz kartê **Aktywacja**. Aby uzyskaæ dostêp do arkuszy w³aœciwoœci, mo¿esz równie¿ zaznaczyæ element, a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **W³aœciwoœci**, albo klikn¹æ przycisk **W³aœciwoœci** z paska narzêdzi MTS.
3. Kliknij kartê **Aktywacja** i okreœl odpowiedni¹ w³aœciwoœæ aktywacji.
4. Kliknij przycisk **OK**, aby powróciæ do programu MTS Explorer.

Zobacz te¿

Ustawianie w³aœciwoœci pakietu MTS, Ustawianie w³aœciwoœci transakcji MTS, Ustawianie poziomów uwierzytelnienia MTS, Blokowanie pakietu MTS

Ustawianie w³aœciwoœci transakcji MTS

U¿ytkownik powinien okreœliæ, czy jego aplikacja ma obs³ugiwaæ transakcjê w czasie projektowania, a nastêpnie w trakcie rozmieszczania u¿yæ programu Microsoft Transaction Server Explorer w celu ustawienia w³aœciwoœci transakcji sk³adnika. Sposób obs³ugi transakcji przez sk³adnik okreœla siê na arkuszu w³aœciwoœci za pomoc¹ karty **Transakcja**. Mo¿na wyró¿niæ nastêpuj¹ce ustawienia w³aœciwoœci transakcji sk³adnika:

- **Wymaga transakcji** Ustawienie powoduje, ¿e obiekty sk³adnika musz¹ zostaæ wykonane w zakresie transakcji. Kiedy jest tworzony nowy obiekt, kontekst tego obiektu dziedziczy transakcjê od kontekstu klienta. Jeœli klient nie posiada transakcji, dla obiektu automatycznie jest tworzona nowa transakcja.
- **Wymaga nowej transakcji** Ustawienie powoduje, ¿e obiekty sk³adnika musz¹ zostaæ wykonane w swoich w³asnych transakcjach. Kiedy jest tworzony nowy obiekt, program Transaction Server automatycznie tworzy now¹ transakcjê dla obiektu, niezale¿nie od tego, czy klient posiada transakcjê.
- **Obs³uguje transakcje** Ustawienie powoduje, ¿e obiekty sk³adnika mog¹ byæ wykonane w zakresie transakcji ich klientów. Kiedy jest tworzony nowy obiekt, kontekst tego obiektu dziedziczy transakcjê od kontekstu klienta. Jeœli klient nie posiada transakcji, nowy kontekst jest tworzony bez transakcji.
- **Nie obs³uguje transakcji** Ustawienie powoduje, ¿e obiekty sk³adnika nie bêd¹ wykonywane w zakresie transakcji. Kiedy jest tworzony nowy obiekt, kontekst obiektu jest tworzony bez transakcji, niezale¿nie od tego czy klient posiada transakcjê.

Za ka¿dym razem, kiedy jest tworzona instancja sk³adnika, program MTS sprawdza atrybut transakcji sk³adnika, aby ustaliæ, czy instancja ta ma byæ uruchomiona w ramach transakcji.

Nie wszystkie sk³adniki obs³uguj¹ przetwarzanie transakcji. Jeœli dany sk³adnik nie mo¿e korzystaæ z przetwarzania transakcji, nale¿y sprawdziæ, czy atrybut transakcji sk³adnika (ustawiany na karcie **Transakcja** sk³adnika) jest ustawiony jako **Nie obs³uguje transakcji**.

Warto zauwa¿yæ, ¿e po *zablokowaniu* sk³adników pakietu modyfikacja atrybutów transakcji jest niemo¿liwa. Wiêcej informacji na ten temat mo¿na znaleŸæ pod has³em Blokowanie pakietu.

Aby ustawiæ w³aœciwoœci transakcji pakietu

1. Zaznacz sk³adnik, który chcesz skonfigurowaæ.
2. W menu **Akcja** kliknij polecenie **W³aœciwoœci** i zaznacz kartê **Transakcja**. Aby uzyskaæ dostêp do arkuszy w³aœciwoœci, mo¿esz równie¿ zaznaczyæ element, a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **W³aœciwoœci**, albo klikn¹æ przycisk **W³aœciwoœci** z paska narzêdzi MTS.
3. Kliknij odpowiednie pole atrybutu transakcji.
4. Kliknij przycisk **OK**.

Przegl¹d informacji na temat rozpowszechniania transakcji oraz metod monitorowania transakcji i zarz¹dzania transakcjami mo¿na znaleŸæ pod has³em Zarz¹dzanie transakcjami MTS.

Zobacz te¿

Ustawianie w³aœciwoœci pakietu MTS, Ustawianie w³aœciwoœci aktywacji MTS, Ustawianie poziomów uwierzytelnienia MTS, Blokowanie pakietu MTS

Ustawianie poziomów uwierzytelnienia MTS

Poziom uwierzytelnienia aplikacji okreœla poziom zabezpieczeñ stosowanych przy uwierzytelnianiu ¿¹dañ klientów. Jeœli klienci danej aplikacji nie musz¹ byæ uwierzytelniani, na karcie **Zabezpieczenia** arkusza w³aœciwoœci pakietu nale¿y wybraæ opcjê **Anonimowe**. Aby przeprowadzaæ uwierzytelnianie klientów, nale¿y wybraæ opcjê **Personifikuj**. Proces serwera ustawi wówczas poziom uwierzytelniania zgodnie ze wskazaniem oraz wymusi personifikacjê. Na uwierzytelnianie nie bêdzie mia³ wp³ywu rozpowszechniany model COM (DCOM).

Jeœli u¿ytkownik nie ma pe³nej wiedzy na temat poziomów uwierzytelnienia modelu DCOM, zaleca siê pozostawienie domyœlnego ustawienia uwierzytelnienia pakietu, czyli opcji **Pakiet**.

W poni¿szej tabeli zestawiono ró¿ne ustawienia uwierzytelniania modelu DCOM:

Poziom	Opis
Brak	W trakcie komunikacji miêdzy danym pakietem a innym pakietem lub aplikacj¹ klienta nie wystêpuje ¿adne sprawdzanie zabezpieczeñ.
Po³¹cz siê	Sprawdzanie zabezbieczeñ wystêpuje tylko przy po³¹czeniu inicjuj¹cym.
Wywo³aj	Sprawdzanie zabezbieczenia wystêpuj¹cego przy ka¿dym wywo³aniu przez ca³y czas po³¹czenia.
Pakiet	To¿samoœæ nadawcy jest szyfrowana.
Integralnoœæ pakietów	To¿samoœæ i podpis nadawcy s¹ szyfrowane, aby nie dopuœciæ do zmiany pakietów podczas przekazywania.
Poufnoœæ pakietów	Aby zapewniæ maksimum bezpieczeñstwa, szyfrowany jest ca³y pakiet, w tym dane oraz to¿samoœæ i podpis nadawcy.

Aby ustawiæ poziomy uwierzytelnienia dla komputera

1. W programie MTS Explorer zaznacz pakiet, który chcesz skonfigurowaæ.
2. W menu **Akcja** kliknij polecenie **W³aœciwoœci** i zaznacz kartê **Zabezpieczenia**. Aby uzyskaæ dostêp do arkuszy w³aœciwoœci, mo¿esz równie¿ zaznaczyæ pakiet, a nastêpnie: albo klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **W³aœciwoœci**, albo klikn¹æ przycisk **W³aœciwoœci** z paska narzêdzi MTS.
3. Zaznacz poziom uwierzytelnienia wywo³añ, który ma byæ skonfigurowany dla tego pakietu.
4. Kliknij przycisk **OK**.

Zobacz te¿

Ustawianie w³aœciwoœci pakietu MTS, Ustawianie w³aœciwoœci aktywacji MTS, Ustawianie w³aœciwoœci transakcji MTS, Blokowanie pakietu MTS

Blokowanie pakietu MTS

Po zaprojektowaniu, zainstalowaniu lub rozmieszczeniu aplikacji warto rozwa¿yæ mo¿liwoœæ zablokowania pakietu, czyli uniemo¿liwienia zmiany konfiguracji sk³adników. Pakiet mo¿na zablokowaæ przed wyeksportowaniem aplikacji serwera do innego komputera obs³uguj¹cego mechanizm MTS (zarz¹dzanego przez system lub administratora sieci Web) lub przed rozpowszechnianiem aplikacji serwera wœród klientów.

Pakiet mo¿na zablokowaæ na wypadek:

- Zmian

  Ustawienie powoduje, ¿e administratorzy nie mog¹ modyfikowaæ konfiguracji pakietu bez uprzedniego wy³¹czenia blokady.
- Usuniêæ

  Ustawienie powoduje, ¿e administratorzy nie mog¹ usun¹æ pakietu bez uprzedniego wy³¹czenia blokady.

Aby zablokowaæ pakiet

1. Po zakoñczeniu konfigurowania pakietu na arkuszu w³aœciwoœci pakietu zaznacz kartê **Zaawansowane**.
2. W sekcji **Uprawnienia** arkusza **Zaawansowane** zaznacz albo opcjê **Wy³¹cz usuwanie** (co uniemo¿liwi u¿ytkownikom usuniêcie pakietu), albo opcjê **Wy³¹cz zmiany** (co uniemo¿liwi u¿ytkownikom wprowadzanie zmian do w³aœciwoœci pakietu).

Aby odblokowaæ pakiet, nale¿y wyczyœciæ pola wyboru na arkuszu w³aœciwoœci **Zaawansowane**.

Zobacz te¿

Ustawianie w³aœciwoœci pakietu MTS, Ustawianie w³aœciwoœci aktywacji MTS, Ustawianie w³aœciwoœci transakcji MTS, Ustawianie poziomów uwierzytelnienia MTS

Rozpowszechnianie pakietów MTS

Program Microsoft Transaction Server Explorer pozwala rozpowszechniaæ pakiety zarówno wœród klientów korzystaj¹cych z programu MTS, jak i wœród klientów nie korzystaj¹cych z niego. Jeœli zarówno komputer klienta, jak i komputer serwera korzystaj¹ z programu MTS, mo¿na za pomoc¹ programu MTS Explorer tworzyæ pliki wykonywalne aplikacji lub za pomoc¹ folderu "Sk³adniki zdalne" przekazywaæ i œci¹gaæ sk³adniki. Jeœli komputer klienta nie korzysta z programu MTS, nale¿y za pomoc¹ programu MTS Explorer wygenerowaæ pliki wykonywalne aplikacji. Pliki te powoduj¹ automatyczn¹ instalacjê i konfiguracjê klientów tak, aby mogli oni uzyskiwaæ dostêp do zdalnych aplikacji serwera MTS w ramach modelu DCOM (rozpowszechnianego modelu COM).

W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Praca ze zdalnymi komputerami MTS

Eksport pakietów MTS

Generowanie plików wykonywalnych MTS

Zobacz te¿

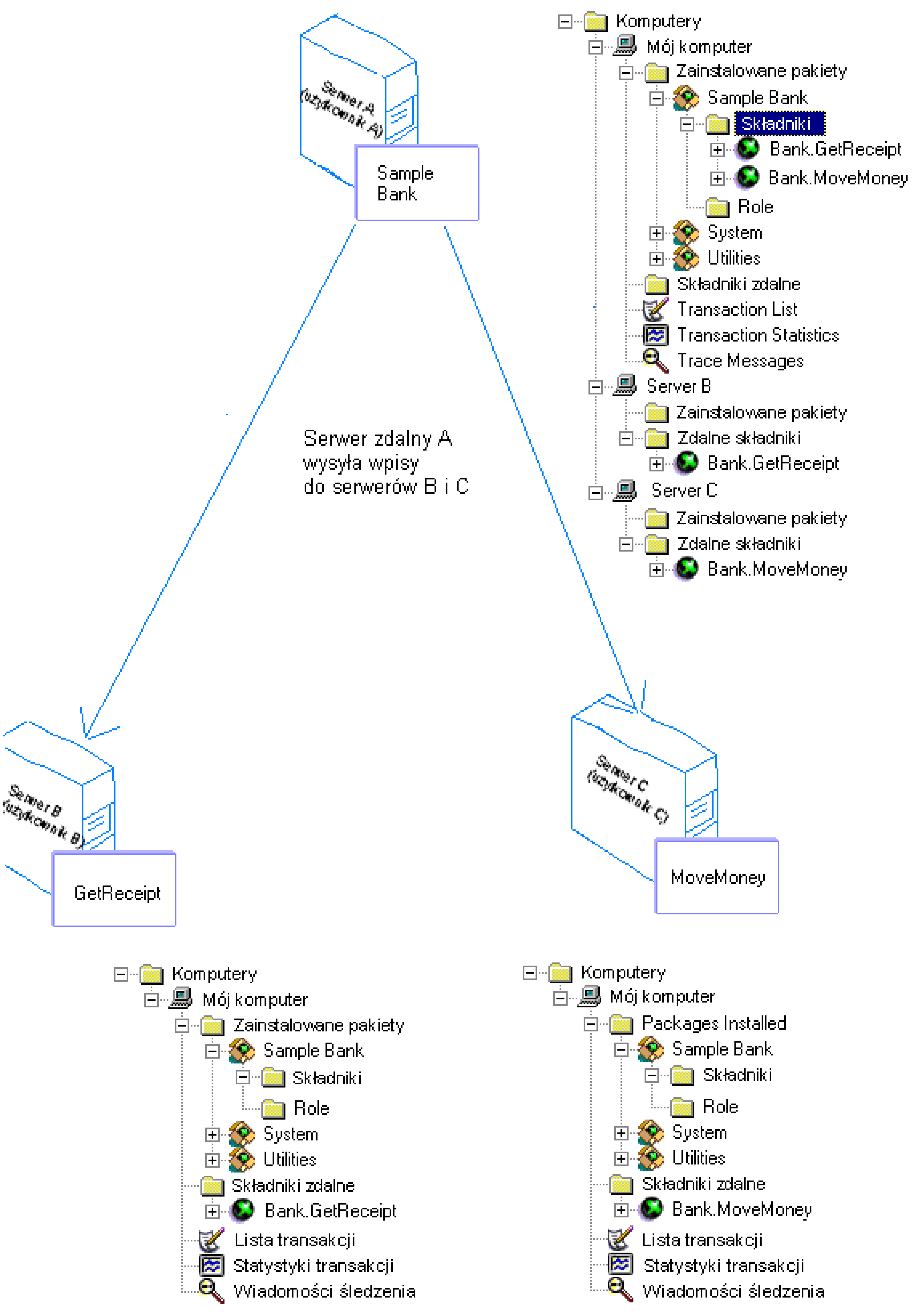
Folder sk³adników zdalnych MTS

Praca ze zdalnymi komputerami MTS

Jeœli zarówno komputer klienta, jak i komputer serwera korzystaj¹ z programu MTS, pakiet mo¿na rozpowszechniaæ "œci¹gaj¹c" i "przekazuj¹c" sk³adniki miêdzy komputerami. Ponadto mo¿na wygenerowaæ plik wykonywalny aplikacji za pomoc¹ narzêdzia do tworzenia plików wykonywalnych aplikacji programu Microsoft Transaction Server Explorer. Wiêcej informacji na temat wykorzystania programu MTS Explorer do generowania plików wykonywalnych aplikacji mo¿na znaleŸæ pod has³em Generowanie plików wykonywalnych MTS.

Przekazywanie sk³adników oznacza tworzenie wpisów dla sk³adników zdalnych na komputerach zdalnych. Jeœli wpisy dla sk³adników zdalnych zostan¹ utworzone, nale¿y dodaæ je do w³asnego folderu "Sk³adniki zdalne" na komputerze lokalnym (œci¹gn¹æ sk³adniki).

Poni¿szy diagram pokazuje, w jaki sposób przekazywanie i œci¹ganie sk³adników wp³ywa na konfiguracjê komputera zdalnego korzystaj¹cego z programu MTS.


Serwer A
(użytkownik A)
Sample
Bank
Komputery
Mój komputer
Zainstalowane pakiety
Sample Bank
Składniki
Bank.GetReceipt
Bank.MoveMoney
Role
System
Utilities
Składniki zdalne
Transaction List
Transaction Statistics
Trace Messages
Server B
Zainstalowane pakiety
Zdalne składniki
Bank.GetReceipt
Server C
Zainstalowane pakiety
Zdalne składniki
Bank.MoveMoney
Serwer zdalny A
wysyła wpisy
do serwerów B i C
Serwer B
(użytkownik B)
GetReceipt
Serwer C
(użytkownik C)
MoveMoney
Komputery
Mój komputer
Zainstalowane pakiety
Sample Bank
Składniki
Role
System
Utilities
Składniki zdalne
Bank.GetReceipt
Lista transakcji
Statystyki transakcji
Wiadomości śledzenia
Komputery
Mój komputer
Packages Installed
Sample Bank
Składniki
Role
System
Utilities
Składniki zdalne
Bank.MoveMoney
Lista transakcji
Statystyki transakcji
Wiadomości śledzenia

W przedstawionym diagramie program MTS zainstalowano na wszystkich trzech komputerach. Sk³adniki GetReceipt i MoveMoney zainstalowano w pakiecie na komputerze u¿ytkownika (Serwer A). Kiedy sk³adniki GetReceipt i MoveMoney zostan¹ dodane do folderów "Zdalne sk³adniki" Serwera B i Serwera C, zdarz¹ siê dwie rzeczy. Po pierwsze, odpowiednie pliki DLL zostan¹ skopiowane za poœrednictwem sieci na Serwer B i Serwer C. Pliki te zostan¹ skopiowane do podkatalogu o tej samej nazwie co pakiet na serwerze A. Podkatalog ten zostanie utworzony w katalogu \<*Microsoft Transaction Server Install directory*>\Remote (na przyk³ad, C:\Program Files\MTx\Remote\Sample Bank). Po drugie, rejestry systemowe Serwera B i Serwera C zostan¹ uaktualnione zgodnie z informacjami z rejestru systemowego Serwera A.

Sk³adniki mo¿na przekazywaæ i œci¹gaæ tylko wówczas, jeœli w celu przechowywania i dostarczania bibliotek typów i bibliotek DLL okreœlono wspó³u¿ytkowany katalog sieciowy. (O ile pliki sk³adników znajduj¹ siê w jednym z folderów lub podfolderów katalogu wspó³u¿ytkowanego, mo¿na wybraæ dowolny katalog wspó³u¿ytkowany.) Program MTS Explorer automatycznie umieœci na serwerach dostêpne sieciowe katalogi wspó³u¿ytkowane. Na danym serwerze mo¿e istnieæ wiele katalogów wspó³u¿ytkowanych, zapewniaj¹cych dostêp do ró¿nych zbiorów plików sk³adników.

Ponadto nale¿y sprawdziæ, czy:

- Zalogowano siê przy u¿yciu konta systemu Windows NT, które jest cz³onkiem roli Administrator pakietu systemowego na komputerze docelowym.
- Identyfikator pakietu systemowego komputera docelowego wskazuje na konto systemu NT, które jest cz³onkiem roli Odczyt u¿ywanego pakietu systemowego.
- Dla pakietów systemowych obu komputerów w³¹czono zabezpieczenie. Wiêcej informacji na ten temat mo¿na znaleŸæ pod has³em W³¹czanie zabezpieczeñ pakietu MTS.

Aby móc przekazywaæ sk³adniki, nale¿y po pierwsze dodaæ odpowiedni komputer lub komputery do u¿ywanego programu MTS Explorer, a nastêpnie dodaæ w³asne sk³adniki do folderu "Zdalne sk³adniki" komputera zdalnego. Wiêcej informacji o dodawaniu komputerów do programu MTS Explorer mo¿na znaleŸæ pod has³em Konfigurowanie serwera MTSr.

Nastêpnie nale¿y dodaæ sk³adniki do folderu "Komputer zdalny" komputera zdalnego.

Aby dodaæ sk³adniki do folderu "Zdalne sk³adniki" komputera zdalnego

1. Dodaj komputer zdalny, zaznaczaj¹c folder "Komputery", a nastêpnie klikaj¹c polecenie **Nowy** z menu **Akcja**.
2. Wpisz nazwê dodawanego komputera, a nastêpnie kliknij przycisk **OK**. Jeœli nie znasz nazwy komputera, skorzystaj z przycisku **Przegl¹daj**.
3. W programie MTS Explorer otwórz folder "Zdalne sk³adniki" z pamiêci komputera, do którego chcesz dodaæ sk³adniki zdalne.
4. Kliknij polecenie **Nowy** z menu **Akcja**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy, a nastêpnie wybraæ z menu polecenia **Nowy** i **Sk³adnik**.
5. W wyœwietlonym oknie dialogowym zaznacz komputer zdalny i pakiet zawieraj¹cy sk³adnik, który chcesz wywo³aæ zdalnie.
6. Na liœcie **Dostêpne sk³adniki** zaznacz sk³adnik, który chcesz wywo³aæ zdalnie, a nastêpnie kliknij strza³kê "w dó³" (Dodawanie). W wyniku powy¿szych czynnoœci sk³adnik oraz komputer, na którym on rezyduje, zostan¹ dodane do pola **Sk³adniki do skonfigurowania**. Po klikniêciu pola wyboru **Szczegó³y** w polu **Sk³adniki do skonfigurowania** zostan¹ wyœwietlone: komputer zdalny, pakiet oraz œcie¿ki dostêpu do bibliotek DLL.
7. Kliknij przycisk **OK**.

Notka Jeœli dodajesz sk³adniki przechowywane w pamiêci komputera zdalnego,

wymagane pliki znajduj¹ siê w katalogu *install directory*\remote. (Domyœlnym katalogiem instalacyjnym programu MTS jest \Program Files\MTS.)

Zobacz te¿

Folder sk³adników zdalnych MTS

Eksport pakietów MTS

Operacja eksportowania pozwala kopiowaæ pakiety z jednego komputera MTS do pamiêci innego. Na przyk³ad za pomoc¹ programu Microsoft Transaction Server Explorer pakiet mo¿na wyeksportowaæ z serwera, na którym pakiet utworzono do innego serwera, na którym bêdzie on testowany.

Aby wyeksportowaæ pakiet:

1. W lewym okienku programu MTS Explorer zaznacz pakiet, który chcesz wyeksportowaæ.
2. W menu **Akcja** kliknij polecenie **Eksportuj**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy i zaznaczyæ polecenie **Eksportuj**.
3. W oknie dialogowym **Eksportuj pakiet** podaj lub odszukaj plik pakietu do utworzenia. Pliki sk³adników zostan¹ skopiowane do tego samego katalogu, w którym znajduje siê plik pakietu.
4. Jeœli do pakietu chcesz do³¹czyæ jakiekolwiek role skojarzone z nim, kliknij pole wyboru **Zapisz identyfikatory u¿ytkowników Windows NT skojarzone z rolami**.
5. Kliknij przycisk **Eksportuj**.

Podczas eksportowania pakietu program Microsoft Transaction Server tworzy plik pakietu (o rozszerzeniu .pak) zawieraj¹cy informacje o sk³adnikach i rolach (jeœli takie istniej¹) wchodz¹cych w sk³ad pierwotnego pakietu oraz kopiuje skojarzone z pakietem pliki sk³adników (biblioteki do³¹czane dynamicznie (DLL), biblioteki typów oraz wszystkie biblioteki DLL typu proxy/stub) do tego samego katalogu, w którym utworzono plik pakietu. Kopiowane s¹ tylko biblioteki DLL sk³adników. Blokady zabezpieczaj¹ce przed zmianami i usuwaniem pakietu s¹ eksportowane wraz z nim.

Wa¿ne Jeœli identyfikator klasy (CLSID), identyfikator biblioteki (TypeLibId) lub identyfikator interfejsu (IID) ulegn¹ zmianie po wyeksportowaniu, pakiet nale¿y eksportowaæ ponownie.

Zobacz te¿

Budowanie pakietów MTS w celu eksportu, Blokowanie pakietu MTS, Mapowanie ról MTS na u¿ytkowników i grupy

Generowanie plików wykonywalnych MTS

Program MTS Explorer pozwala generowaæ pliki wykonywalne aplikacji, które s³u¿¹ do instalacji i konfigurowania komputera klienta tak, aby zapewniæ mu dostêp do aplikacji serwera zdalnego. Komputer klienta, na którym jest instalowany plik wykonywalny, musi zapewniaæ obs³ugê modelu DCOM (z ang. distributed COM), ale nie musi zawieraæ ¿adnych innych plików serwera MTS poza plikiem wykonywalnym zapewniaj¹cym dostêp do aplikacji serwera zdalnego MTS.

Narzêdzie do tworzenia plików wykonywalnych aplikacji jest elementem funkcji eksportowania pakietów programu MTS. Narzêdzie to pozwala w sposób automatyczny generowaæ pliki wykonywalne aplikacji, które instaluj¹ aplikacje klientów i konfiguruj¹ komputery klientów tak, aby zapewniæ im dostêp do aplikacji serwera na zdalnym serwerze MTS. Pliki wykonywalne generowane przez narzêdzie domyœlnie powoduj¹ takie skonfigurowanie komputera klienta, aby mia³ one dostêp do serwera, na którym wegenerowano plik wykonywalny.

Aby aplikacje klientów wskazywa³y na inny serwer ni¿ serwer rozmieszczaj¹cy, mo¿na wprowadziæ odpowiednie ustawienia na karcie **Opcje** arkusza w³aœciwoœci komputera. Jeœli przed eksportowaniem pakietu generuj¹cego plik wykonywalny nie zostanie podana inna nazwa komputera serwera, pliki wykonywalne aplikacji utworzone na komputerze lokalnym automatycznie tak skonfiguruj¹ komputery klientów, aby zapewniæ im dostêp do pakietów serwera na komputerze lokalnym.

Aby skonfigurowaæ plik wykonywalny aplikacji klienta w sposób zapewniaj¹cy dostêp do innego komputera serwera ni¿ komputer lokalny:

1. Zaznacz ikonê **Mój Komputer**.
2. Kliknij prawym przyciskiem myszy, a nastêpnie wybierz polecenie **W³aœciwoœci**.
3. Zaznacz kartê **Opcje**, a nastêpnie w polu **Nazwa serwera zdalnego** sekcji **Replikacja** podaj nazwê serwera zdalnego.
4. Kliknij przycisk **OK** i wyeksportuj pakiet, aby utworzyæ plik wykonywalny aplikacji.

Po stronie serwera MTS narzêdzie do tworzenia plików wykonywalnych aplikacji automatycznie tworzy plik wykonywalny dla aplikacji klienta.

Po stronie klienta plik wykonywalny automatyzuje nastêpuj¹ce kroki:

1. Kopiuje siê do katalogu tymczasowego w komputerze klienta lub serwera, a nastêpnie rozpakowuje wszystkie pliki potrzebne klientowi, w tym biblioteki typów i niestandardowe biblioteki DLL typu proxy/stub.
2. Przesy³a biblioteki typów i biblioteki DLL typu proxy-stub dla aplikacji serwera do katalogu "Aplikacje zdalne", znajduj¹cego siê w katalogu \Program Files. Wszystkie aplikacje zdalne s¹ przechowywane w katalogu "Aplikacje zdalne". Ka¿da aplikacja zdalna ma swój w³asny katalog o nazwie okreœlonej przez unikatowy identyfikator globalny pakietu (identyfikator GUID).
3. Aktualizuje rejestr systemu o wpisy, które albo pozwalaj¹ korzystaæ klientom z aplikacji serwera w sposób zdalny (w³¹czaj¹c w to informacje zwi¹zane z identyfikatorami aplikacji, klas, interfejsów, bibliotek i programistycznymi) albo umo¿liwiaj¹ uruchamianie aplikacji serwera na komputerze serwera.
4. Rejestruje aplikacjê, dziêki czemu u¿ytkownik mo¿e j¹ w razie potrzeby usun¹æ w Panelu sterowania, za pomoc¹ ikony **Dodaj/Usuñ programy**. Wszystkie aplikacje zdalne s¹ poprzedzane okreœleniem "Aplikacja zdalna", dziêki czemu ³atwo je znaleŸæ na liœcie zainstalowanych sk³adników.
5. Usuwa pliki z tymczasowego katalogu wygenerowanego podczas instalowania aplikacji.

Po uruchomieniu na komputerze klienta plik wykonywalny kopiuje do pamiêci tego komputera wszystkie niezbêdne biblioteki DLL typu proxy-stub i biblioteki typów oraz

ukatualnia rejestr systemowy klienta o informacje wymagane w modelu DCOM, miêdzy innymi o nazwê komputera serwera. Dziêki temu aplikacje klientów mog¹ uzyskiwaæ dostêp do aplikacji serwera zdalnego.

Przed eksportowaniem pakietu tworz¹cego plik wykonywalny nale¿y skonfigurowaæ plik instalacyjny klienta (clients.ini). Instalacjê klienck¹ mo¿na dostosowaæ, do³¹czaj¹c do niej dodatkowe pliki, na przyk³ad pliki wykonywalne klienta, dokumentacjê aplikacji lub proste pliki typu "readme". Na przyk³ad plik clients.ini file, znajduj¹cy siê w podkatalogu \Clients, mo¿na zmodyfikowaæ w sposób pozwalaj¹cy instalowaæ pliki wykonywalne klienta dla kilku ró¿nych aplikacji.

Aby dostosowaæ instalacjê:

1. Otwórz plik clients.ini, znajduj¹cy siê w podkatalogu \clients.
2. Pod nag³ówkiem "Pliki aplikacji klienta"(ang. "Client Application Files") wprowadŸ œcie¿kê dostêpu do katalogu, w którym ma byæ zapisany kod Ÿród³owy instalowany w komputerze klienta. Na przyk³ad:

   Source Path=c:\pgram files\mtx\test\vb bank
3. Pod nag³ówkiem ClientApplicationInstallCommands wprowadŸ nazwy plików, które chcesz zainstalowaæ. Nazwy instalowanych plików umieœæ w potrójnych nawiasach klamrowych. Na przyk³ad:

   1=notepad {{{readme.txt}}}
   2={{{vbbank.ex}}}}
4. Pod nag³ówkiem ClientApplicationSetup okreœl, czy chcesz w³¹czyæ opcjê ExploreApplication instalacji. Jeœli tak, wpisz “Y”, jeœli nie, wpisz “N”. Na przyk³ad poni¿szy wpis powoduje wyœwietlanie aplikacji natychmiast po jej zainstalowaniu, dziêki czemu klient mo¿e utworzyæ skrót do pulpitu:

   ExploreApplication=Y

Po dostosowaniu pliku clients.ini mo¿na przyst¹piæ do eksportowania (za pomoc¹ programu MTS Explorer) pakietu serwera tworz¹cego plik wykonywalny klienta.

Aby utworzyæ plik wykonywalny klienta:

1. Jeœli jeszcze tego nie wykonano, zainstaluj aplikacjê serwera (za pomoc¹ Kreatora pakietów).
2. Jeœli chcesz, aby klienci mieli dostêp do innego serwera ni¿ serwer rozmieszczaj¹cy, na którym utworzono plik wykonywalny, wykonaj nastêpuj¹ce kroki.
   - W lewym okienku programu MTS Explorer zaznacz ikonê **Mój Komputer**.
   - Kliknij prawym przyciskiem myszy, a nastêpnie wybierz polecenie **W³aœciwoœci**.
   - Kliknij kartê **Opcje**. W polu **Nazwa serwera zdalnego** wprowadŸ nazwê serwera, do którego bêd¹ mieli dostêp klienci. Na przyk³ad:

     \\anotherservermachine

1. Wyeksportuj pakiet z serwera, na którym zainstalowano aplikacjê serwera do innego serwera. Podaj now¹ nazwê eksportowanego pakietu (*YourNewFileNam*e), a nastêpnie umieœæ eksportowany pakiet we w³asnym katalogu programu MTS.
2. Odszukaj folder, do którego pakiet ma byæ eksportowany. Na ekranie pojawi siê podkatalog "Klienci" (ang. "Clients") zawieraj¹cy pojedynczy plik o nazwie *YourNewFileName*.exe. Eksport istniej¹cego pakietu powoduje, ¿e podkatalog "Klienci" (ang. "Clients") jest generowany w katalogu docelowym eksportu. Podkatalog ten zawiera pojedynczy plik wykonywalny o nazwie okreœlonej podczas eksportu pakietu. Po uruchomieniu tego pliku na komputerze klienta obs³uguj¹cym model DCOM s¹ instalowane wszystkie informacje zapewniaj¹ce klientom zdalnym dostêp do aplikacji serwera.

Wa¿ne Nie mo¿na uruchamiaæ opisywanego wy¿ej pliku wykonywalnego klienta na

komputerze serwera. Spowoduje to usuniêcie wpisów rejestru niezbêdnych do uruchomienia pakietu serwera. Jeœli u¿ytkownik pope³ni ten b³¹d, powinien usun¹æ aplikacjê w Panelu sterowania, za pomoc¹ arkuszy w³aœciwoœci **Dodaj/Usuñ programy**. Nastêpnie powinien usun¹æ i ponownie zainstalowaæ pakiet (za pomoc¹ programu MTS Explorer).

Rozpowszechnianie pliku wykonywalnego klienta

Program MTS Explorer automatycznie tworzy pakiety i instaluje aplikacje klientów w postaci plików wykonywalnych przeznaczonych do rozpowszechniania. Utworzone pliki wykonywalne mo¿na rozpowszechniaæ nastêpuj¹cymi metodami:

- Wspó³u¿ytkowaæ katalog, z którego klienci mogliby kopiowaæ plik wykonywalny i uruchamiaæ go na swoich komputerach.
- Wys³aæ za³¹cznik poczty elektronicznej tak, aby klienci mogli zapisaæ i uruchomiæ plik wykonywalny na swoich komputerach.
- Do³¹czyæ, za pomoc¹ taga <OBJECT>, plik wykonywalny do skryptu HTML. Tag <OBJECT> powoduje, ¿e jeœli klient zainicjuje na stronie HTML okreœlone zdarzenie (na przyk³ad kliknie mysz¹), przegl¹darka mo¿e pobraæ aplikacjê z podanej lokalizacji magazynu obiektów komputera klienta. Zastosowanie taga <OBJECT> do rozpowszechniania plików wykonywalnych u³atwia uaktualnienie plików wykonywalnych, poniewa¿ przegl¹darka automatycznie sprawdza rejestr klienta w celu ustalenia aktualnej wersji aplikacji. Jeœli istniej¹cy plik wykonywalny siê zdezaktualizuje, przegl¹darka spróbuje pobraæ jego najnowsz¹ wersjê (z magazynu obiektów).
- Za pomoc¹ programu Microsoft System Management Server (SMS) rozpowszechniaæ aplikacjê z jej lokalizacji centralnej do dziesi¹tków i setek komputerów jednoczeœnie. Proszê zauwa¿yæ, ¿e po zainstalowaniu aplikacji na komputerach zdalnych nale¿y zainstalowaæ sam¹ aplikacjê klienta.

Usuwanie pliku wykonywalnego klienta

Klienci mog¹ usuwaæ plik wykonywalny klienta z Panelu sterowania za pomoc¹ ikony **Dodaj/Usuñ programy**. Aplikacje instalowane przez pliki wykonywalne MTS s¹ wyró¿nianie na liœcie "Instaluj/Odinstaluj" przez okreœlenie "Aplikacja zdalna" (ang. "Remote Application"). Aby usun¹æ plik wykonywalny, nale¿y zaznaczyæ odpowiedni¹ aplikacjê i klikn¹æ przycisk **Usuñ**.

Zobacz te¿

Eksport pakietów MTS, Budowanie pakietów MTS w celu eksportu

Instalacja pakietów MTS

Do instalowania pakietów MTS s³u¿y program MTS Explorer. Program umo¿liwia instalacjê i rozmieszczenie pakietów w œrodowisku czasu wykonywania programu MTS. Procedury instalacji i rozmieszczania s¹ œciœle zwi¹zane z czynnoœciami projektowymi, na przyk³ad z ustawieniem odpowiedniej to¿samoœci pakietu.

Prawid³owe rozmieszczanie aplikacji wymaga dok³adnego zrozumienia procesu projektowania pakietu i jego sk³adników. Wyczerpuj¹ce informacje na temat projektowania i budowania sk³adników MTS mo¿na znaleŸæ w podrêczniku *MTS Programmer's Guide*.

Program MTS Explorer pozwala w ³atwy sposób rozmieszczaæ pakiety wstêpnie wbudowane. W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Instalacja pakietów wstêpnie wbudowanych

Uaktualnianie pakietów MTS

W³¹czanie zabezpieczenia pakietu MTS

Ustawianie to¿samoœci pakietu MTS

Dodawanie nowej roli MTS

Mapowanie ról MTS do u¿ytkowników i grup

Zobacz te¿

To¿samoœæ, karta (Pakiet), Folder "Zainstalowane pakiety", Folder "Role", Folder "U¿ytkownicy", Folder "Przynale¿noœæ ról", Zarz¹dzanie u¿ytkownikami dla ról MTS

Instalacja pakietów wstêpnie wbudowanych

Pakiet wbudowany sk³ada siê z pliku pakietu oraz ze skojarzonych z pakietem plików sk³adników (z rozszerzeniami .dll i .tlb). Za pomoc¹ programu MTS Explorer mo¿na instalowaæ i rozmieszczaæ sk³adniki wbudowane (kupione lub uzyskane od niezale¿nych producentów zajmuj¹cych siê ich projektowaniem lub projektantów).

Po zainstalowaniu pakietu wybierana jest jego *minimalna* konfiguracja, która obejmuje: sterowanie dostêpem do obiektów przechowywanych przez program MTS, mapowania u¿ytkowników do ról i ustawienia poziomów zabezpieczeñ pakietu. Wiêcej informacji na temat konfigurowania zabezpieczeñ pakietu mo¿na znaleŸæ pod has³em W³¹czanie zabezpieczenia pakietu MTS.

Aby zainstalowaæ pakiet wbudowany:

1. W lewym okienku programu MTS Explorer zaznacz komputer, na którym chcesz utworzyæ pakiet.
2. Otwórz folder **Zainstalowane pakiety**.
3. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie¿ skorzystaæ z przycisku **Utwórz nowy obiekt** wyœwietlanego na pasku narzêdzi MTS lub klikn¹æ prawym przyciskiem myszy folder **Zainstalowane pakiety**, a nastêpnie wybraæ polecenia **Nowy** i **Pakiet**.
4. Kliknij przycisk **Zainstaluj wbudowane pakiety**.
5. Za pomoc¹ przycisku **Dodaj** z okna dialogowego **Wybierz pliki pakietów** odszukaj w sieci dostêpne pliki pakietów. Zaznacz plik pakietu (.pak), kliknij przycisk **Otwórz**, a nastêpnie przycisk **Dalej**. Jednoczeœnie mo¿esz zainstalowaæ kilka pakietów. Znajduj¹ce siê w pakiecie pliki sk³adników musz¹ byæ zapisane w tym samym katalogu, w którym zapisano plik pakietu.
6. W oknie dialogowym **Ustaw to¿samoœæ pakietu** okreœl to¿samoœæ pakietu, a nastêpnie kliknij przycisk **Dalej**. Domyœlnym ustawieniem to¿samoœci jest **U¿ytkownik interakcyjny**. U¿ytkownik interaktywny oznacza u¿ytkownika zalogowanego na koncie komputera wyposa¿onego w system Windows NT, na którym dzia³a pakiet. Mo¿na wybraæ innego u¿ytkownika, zaznaczaj¹c opcjê **Ten u¿ytkownik** i podaj¹c szczegó³owe informacje o konkretnej grupie lub u¿ytkowniku systemu.
7. W oknie dialogowym **Opcje instalacji** podaj katalog instalacji. Do katalogu tego s¹ kopiowane pliki sk³adników zapisane w katalogu pliku pakietu. Mo¿esz zaakaceptowaæ proponowany katalog domyœlny lub lub za pomoc¹ przycisku **Przegl¹daj** odszukaæ inn¹ lokalizacjê.
8. Jeœli instalowany plik pakietu zawiera zdefiniowanych u¿ytkowników systemu Windows NT, dostêpna bêdzie opcja **Dodaj u¿ytkowników NT zapisanych w pliku pakietu**. Klikniêcie tego pola spowoduje dodanie tych u¿ytkowników do nowego pakietu.
9. Kliknij przycisk **Zakoñcz**. W prawym okienku hierarchii programu MTS Explorer zostanie pokazany nowy pakiet. Jeœli zainstalowano kilka pakietów, opcje wybrane w kroku 6 i 7 zostan¹ zastosowane do wszystkich.

W systemie Windows NT pakiet wstêpnie wbudowany mo¿na równie¿ zainstalowaæ, otwieraj¹c folder **Pakiety** komputera, a nastêpnie przeci¹gaj¹c plik pakietu z okna programu Eksplorator Windows NT do prawego okienka programu MTS Explorer.

Zobacz te¿

Folder "Zainstalowane pakiety"

Uaktualnianie pakietów MTS

Aby uaktualniæ pakiet MTS, nale¿y usun¹æ jego poprzedni¹ wersjê i zainstalowaæ wersjê uakualnion¹.

Aby usun¹æ pakiet:

1. W lewym okienku programu MTS Explorer zaznacz komputer, na którym chcesz usun¹æ pakiet.
2. Otwórz folder **Zainstalowane pakiety**, aby wyœwietliæ wszystkie pakiety dostêpne na tym komputera.
3. Zaznacz pakiet, który chcesz usun¹æ.
4. W menu **Akcja** kliknij polecenie **Usuñ**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **Usuñ**.
5. Kliknij przycisk **Tak**, aby usun¹æ pakiet.

Usuniêcie pakietu powoduje usuniêcie wszystkich sk³adników zawartych w pakiecie. Pakiet mo¿na usun¹æ, zaznaczaj¹c go i naciskaj¹c klawisz DELETE.

Sposób instalacji pakietu uaktualnionego opisano pod has³em Instalacja wbudowanych pakietów MTS.

Zobacz te¿

Folder "Zainstalowane pakiety"

W³¹czanie zabezpieczenia pakietu MTS

Program MTS oferuje dwa rodzaje zabezpieczeñ pakietu:

- Zabezpieczeniae programowalne

  Dostarczane s¹ interfejsy, za pomoc¹ których mo¿na tworzyæ niestandardowe zabezpieczenie w ramach kodu aplikacji. Wiêcej informacji na temat korzystania z zabezpieczeñ programowalnych mo¿na znaleŸæ w podrêczniku *MTS Programmer's Guide*.
- Zabezpieczenie deklaratywne

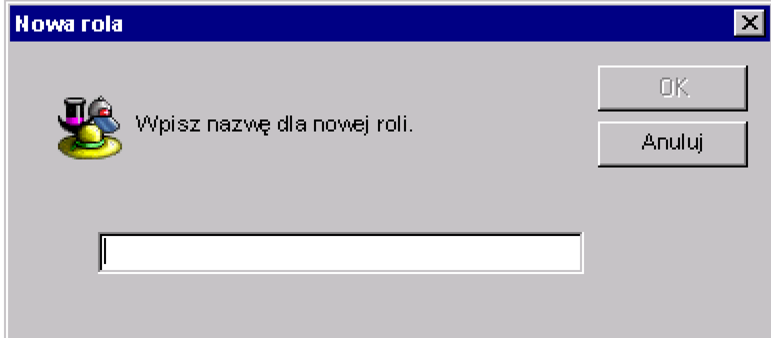
  Zabezpieczenia s¹ tworzone przez definiowanie ról i przypisywanie (za pomoc¹ programu MTS Explorer) do ról u¿ytkowników i grup u¿ytkowników systemu Windows NT.

Wa¿ne Pakiety bibliotek nie obs³uguj¹ sprawdzania ról. W celu w³¹czenia zabezpieczenia nale¿y zmieniæ ustawienie aktywacji na pakiet serwera. Wiêcej informacji o bibliotekach i pakietach serwera mo¿na znaleŸæ pod has³em Ustawianie w³aœciwoœci aktywacji MTS.

Administratorzy chroni¹ pakiety za pomoc¹ zabezpieczenia deklaratywnego, w zwi¹zku z czym mog¹ je uruchamiaæ tylko klienci z uprawnieniami dostêpu. Prawa dostêpu s¹ udzielane za pomoc¹ programu MTS, przy u¿yciu ról MTS lub kont u¿ytkowników i grup systemu Windows NT. Proszê zauwa¿yæ, ¿e poniewa¿ zabezpieczenia deklaratywne przy sprawdzaniu uwierzytelnieñ wymagaj¹ kont systemu Windows NT, nie mo¿na ich u¿ywaæ dla pakietów uruchamianych w systemie Windows 95.

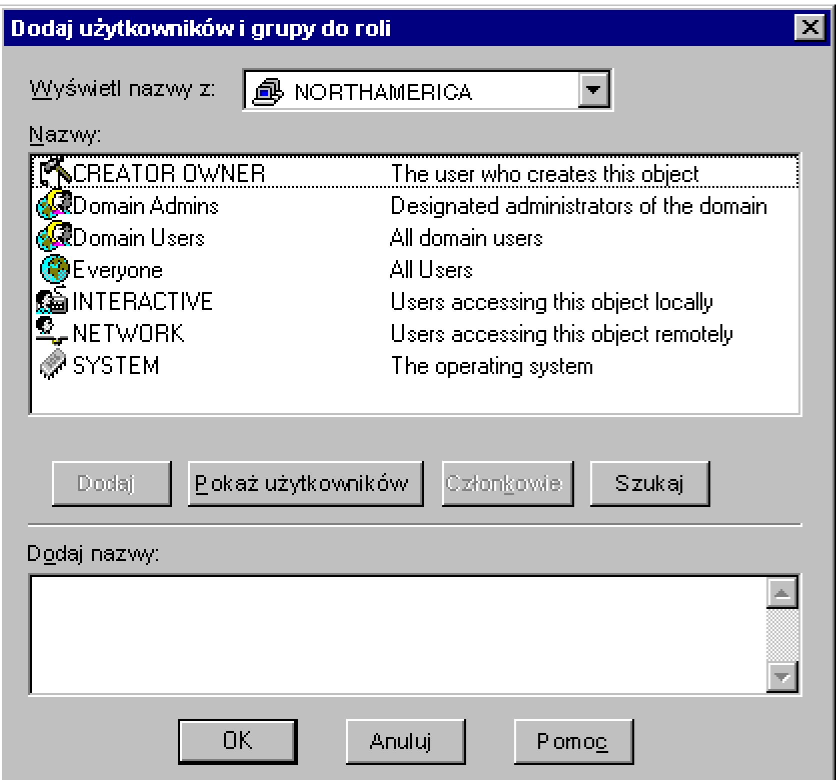
Aby okreœliæ zabezpieczenie deklaratywne pakietu, nale¿y wykonaæ nastêpuj¹ce kroki:

1. W oknie dialogowym **Nowa rola** zdefiniowaæ role na poziomie pakietu.



   Operacje dodawania nowych ról opisano w temacie Dodawanie nowej roli MTS.

1. Za pomoc¹ okna dialogowego **Dodaj nowych u¿ytkowników do ról** zamapowaæ u¿ytkowników do ról. Wywo³anie pakietu bez prawid³owych u¿ytkowników w jakiejkolwiek roli nie jest mo¿liwe.



Informacje na temat dodawania u¿ytkowników i grup do ról mo¿na znaleŸæ pod has³em Mapowanie ról MTS do u¿ytkowników i grup.

1. Jeœli konieczne jest ograniczenie dostêpu do konkretnego sk³adnika lub interfejsu, przypisaæ zdefiniowan¹ rolê do folderu "Przynale¿noœæ roli" danego sk³adnika lub interfejsu.
2. Na karcie **Zabezpieczenia** na arkuszu w³aœciwoœci pakietu w³¹czyæ zabezpieczenia. Niniejszy temat zawiera opis w³¹czania sprawdzania upowa¿nieñ.

Jeœli przed w³¹czeniem zabezpieczenia pakietu systemowego nie zamapowano aktualnie u¿ywanego konta u¿ytkownika do roli administratora, to nie bêdzie mo¿na uzyskaæ dostêpu do funkcji programu MTS Explorer pozwalaj¹cych modyfikowaæ konfiguracjê (na przyk³ad dodawania u¿ytkowników do ról). Jeœli wyst¹pi taka ewentualnoœæ, nale¿y zalogowaæ siê jako u¿ytkownik zamapowany do roli administratora. Aby uchroniæ administratorów przed mo¿liwoœci¹ zablokowania przez pakiet systemowy, program MTS Explorer wyœwietla komunikaty o b³êdach przy ka¿dej próbie:

- W³¹czenia zabezpieczenia pakietu systemowego, kiedy do roli administratora nie zamapowano ¿adnych u¿ytkowników.
- Usuniêcia ostatniego u¿ytkownika z roli administratora, kiedy dzia³a zabezpieczenie pakietu systemowego.

Uwaga Jeœli program MTS zainstalowano na serwerze, którego rolê okreœlono jako kontroler podstawowej lub zapasowej domeny, to pakietami mog¹ zarz¹dzaæ jedynie u¿ytkownicy bêd¹cy administratorami domeny.

Jeœli nie w³¹czono zabezpieczenia pakietu, program MTS nie bêdzie sprawdzaæ ról dla sk³adnika lub interefejsu. Jeœli nie w³¹czono zabezpieczenia sk³adnika, program MTS nie bêdzie sprawdzaæ ról dla interfejsu sk³adnika.

Procedurê przypisywania ról do folderu "Przynale¿noœæ ról" opisano w temacie Dodawanie nowej roli MTS.

Uwaga Wy³¹czenie zabezpieczenia deklaratywnego pakietu lub pojedynczych sk³adników jest u¿yteczne podczas debugowania pakietu.

Rozwa¿my przyk³ad procedury ograniczenia praw dostêpu do pakietu "Magazyn". Za³ó¿my, ¿e administrator systemu zamierza pozostawiæ dostêp do pakietu "Inventory" tylko pracownikom dzia³u sprzeda¿y . W tym celu musi zaznaczyæ dla pakietu "Inventory" folder "Rola", klikn¹æ polecenie **Nowy** z menu **Akcja**, po czym wpisaæ nazwê nowej roli, na przyk³ad "Sprzeda¿". Nastêpnie powinien zaznaczyæ folder "U¿ytkownicy", jeszcze raz klikn¹æ polecenie **Nowy** z menu **Akcja** i wpisaæ nazwê konta grupy systemu Windows NT obejmuj¹cej pracowników dzia³u sprzeda¿y. W dalszej kolejnoœci powinien dodaæ rolê "Sprzeda¿" do folderów "Przynale¿noœæ ról" wszystkich sk³adników. W tym momencie tylko pracownicy dzia³u sprzeda¿y bêd¹ mieli dostêp do pakietu "Magazyn". Ostatecznie, administrator powinien zaznaczyæ pakiet, wywo³aæ arkusz w³aœciwoœci, wybraæ na nim kartê **Zabezpieczenia** i zaznaczyæ pole wyboru **W³¹cz sprawdzanie upowa¿nienia**. Zostan¹ wówczas w³¹czone nowe ustawienia zabezpieczenia pakietu.

Poprawne wykonanie procedury ograniczania praw dostêpu do sk³adników pakietu wymaga wiedzy na temat wzajemnego wywo³ywania siê sk³adników pakietu. Jeœli sk³adnik jest wywo³ywany bezpoœrednio przez klienta podstawowego, program MTS sprawdza zwi¹zane z nim role. Jeœli pewien sk³adnik wywo³uje inny sk³adnik z tego samego pakietu, program MTS nie sprawdza ról (w ramach tego samego pakietu obowi¹zuje bowiem "zasada wzajemnego zaufania" sk³adników).

Za³ó¿my, ¿e pewien u¿ytkownik zamierza tak skonfigurowaæ role, aby klient mia³ uprawnienia do wywo³ywania sk³adnika CheckInventory, a nie mia³ uprawnieñ do bezpoœredniego wywo³ywania sk³adnika Backorder. Obydwa sk³adniki, CheckInventory i Backorder, znajduj¹ siê w pakiecie "Inventory". W pierwszej kolejnoœci nale¿y ustawiæ odpowiedni¹ rolê klienck¹ dla sk³adnika CheckInventory. Nastêpnie nale¿y upewniæ siê, czy sk³adnik Backorder nie posiada ¿adnych ról, które mog³yby odnosiæ siê do klienta (to¿samoœci klienta). Poniewa¿ obydwa sk³adniki znajduj¹ siê we wspólnym pakiecie, dlatego te¿ podczas wywo³ania sk³adnika Backorder przez sk³adnik CheckInventory nie bêdzie wykonywane ¿adne sprawdzanie ról.

Tym niemniej, jeœli spe³nione bêd¹ poni¿sze warunki, sk³adnik CheckInventory mo¿e w imieniu klienta wywo³aæ sk³adnik Backorder:

- To¿samoœæ klienta jest zamapowana do odpowiedniej roli sk³adnika CheckInventory.
- Spe³nione s¹ jakiekolwiek wymagania dotycz¹ce zabezpieczenia programowalnego.

Opisany wy¿ej mechanizm pozwala tworzyæ pakiety zawieraj¹ce sk³adniki "ufaj¹ce sobie nawzajem" , ograniczaj¹c jednoczeœnie uprawnienia do ich wybierania.

Aby w³¹czyæ sprawdzanie ról dla pierwotnych obiektów wywo³uj¹cych, które wywo³uj¹ sk³adnik Backorder bezpoœrednio, nale¿y zaznczyæ folder "Przynale¿noœæ ról" dla sk³adnika Backorder, klikn¹æ polecenie **Nowy** z menu **Akcja**, po czym wybraæ rolê **Sprzeda¿**. W ten sposób rola **Sprzeda¿** (wraz z mapowanymi u¿ytkownikami) zostanie przypisana do sk³adnika Backorder, w zwi¹zku z czym sk³adnik Backorder bêd¹ mogli uruchamiaæ jedynie pracownicy dzia³u sprzeda¿y. Aby uaktywniæ nowe ustawienie zabezpieczenia, nale¿y zaznaczyæ pole wyboru **W³¹cz sprawdzanie upowa¿nienia** zarówno dla pakietu "Magazyn", jak i dla sk³adnika Backorder.

Wiêcej informacji na temat sprawdzania ról mo¿na znaleŸæ w podrêczniku *MTS Programmer's Guide*, w podrozdziale Zabezpieczenia programowalne (ang. Programmatic Security).

Aby w³¹czyæ sprawdzanie to¿samoœci dla zabezpieczenia:

1. Jeœli jeszcze tego nie wykonano, zamapuj swoje konto u¿ytkownika do roli **Administrator** pakietu systemowego.

2 Zaznacz pakiet systemowy i kliknij polecenie **Akcja** z menu **W³aœciwoœci** lub menu wyœwietlanego po klikniêciu prawym przyciskiem myszy.
3 PrzejdŸ do karty zabezpieczenia i zaznacz pole wyboru **W³¹cz sprawdzanie upowa¿nienia** .
4 Zatrzymaj proces serwera pakietu systemowego, zaznaczaj¹c pakiet systemowy, klikaj¹c prawym przyciskiem myszy i wybieraj¹c polecenie **Zamknij**.

   Zamiast wykonywania kroków od 4 do 7, mo¿esz zamkn¹æ wszystkie pakiety serwera jednoczeœnie. W tym celu wybierz ikonê **Mój Komputer**, a nastêpnie kliknij polecenie **Zamknij proces serwera** z menu **Akcja**.
5 Zaznacz pakiet, dla którego chcesz w³¹czyæ zabezpieczenie.
6 PrzejdŸ do karty zabezpieczenia i zaznacz pole wyboru **W³¹cz sprawdzanie upowa¿nienia** .
7 Zatrzymaj proces serwera pakietu systemowego, zaznaczaj¹c ten pakiet, klikaj¹c prawym przyciskiem myszy i wybieraj¹c polecenie **Zamknij**.

Po zainstalowaniu i skonfigurowaniu pakietu na serwerze Ÿród³owym rozmieszczania mo¿na uniemo¿liwiæ przysz³e zmiany konfiguracji sk³adników, blokuj¹c pakiet. Wiêcej informacji na temat blokowania konfiguracji pakietu mo¿na znaleŸæ pod has³em Blokowanie pakietu.

Zobacz te¿

Pakiet systemowy, Folder "Role", Folder "U¿ytkownicy", Folder "Przynale¿noœæ ról", Zarz¹dzanie u¿ytkownikami dla ról MTS, Podrêcznik Microsoft Transaction Server Programmer's Guide

Ustawianie to¿samoœci pakietu MTS

Podczas konfigurowania pakietu (tj. pojedynczego procesu serwera) nale¿y zdecydowaæ siê na jedno z dwóch ustawieñ to¿samoœci:

- U¿ytkownik interaktywny
- Okreœlone konto u¿ytkownika systemu Windows NT

Domyœlnie pakiety s¹ uruchamiane z ustawieniem **U¿ytkownik interakcyjny**.

W wielu scenariuszach rozmieszczania jest po¿¹dane uruchamianie pakietu jako konta u¿ytkownika systemu Windows NT. W takim wypadku mo¿na skonfigurowaæ dostêp do konta jako elementu bazy danych, a nie szczegó³owo, dla konkretnych klientów korzystaj¹cych z pakietu. Przydzielanie uprawnieñ dostêpu do kont, zamiast do pojedynczych klientów zwiêksza skalowalnoœæ aplikacji.

Rozwa¿my przyk³ad pakietu "Ksiêgowoœæ" aktualizuj¹cego bazê danych serwera SQL zawieraj¹c¹ informacje o p³atnoœciach i sprzeda¿y. Tabelê Accounting bazy danych mo¿na skonfigurowaæ tak, aby przydzieliæ prawo odczytu kontu pewnej grupy urzêdników   u¿ytkowników systemu Windows NT. Aby u¿ytkownicy tego konta mogli uruchomiæ pakiet i odczytywaæ dane z tabeli Accounting, nale¿a³oby ustawiæ to¿samoœæ pakietu jako to w³aœnie konto.

Aby ustawiæ to¿samoœæ pakietu jako okreœlone konto u¿ytkownika:

1. Zaznacz pakiet, którego to¿samoœæ chcesz zmieniæ.
2. W menu **Akcja** kliknij polecenie **W³aœciwoœci** i zaznacz kartê **To¿samoœæ**.
3. Zaznacz opcjê **Ten u¿ytkownik**, a nastêpnie podaj: domenê u¿ytkownika poprzedzon¹ odwrotnym ukoœnikiem (\), nazwê u¿ytkownika oraz has³o konta u¿ytkownika w systemie Windows NT.

   Jeœli chcesz za pomoc¹ ustawieñ to¿samoœci pakietu ograniczyæ prawo dostêpu do bazy danych, musisz okreœliæ dla konta u¿ytkownika uprawnienia dostêpu do bazy danych.

Zobacz te¿

Mapowanie ról MTS do u¿ytkowników i grup, W³¹czanie zabezpieczeñ pakietu MTS, Dodawanie nowej roli MTS, To¿samoœæ, karta (Pakiet)

Dodawanie nowej roli MTS

Mimo ¿e nowe role zazwyczaj s¹ dodawane w trakcie tworzenia pakietu, mo¿na je dodawaæ równie¿ do istniej¹cych pakietów. Role reprezentuj¹ zbiór uprawnieñ obowi¹zuj¹cych na poziomie systemu i wymaganych przez konkretn¹ funkcjê aplikacji. Role ustawia siê na poziomie pakietu. Za pomoc¹ programu MTS Explorer do ról mo¿na mapowaæ u¿ytkowników i grupy u¿ytkowników systemu Windows NT.

Aby utworzyæ now¹ rolê:

1. W lewym okienku programu MTS Explorer zaznacz pakiet, który bêdzie zawiera³ rolê.
2. Otwórz folder “Role”.
3. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie zaznaczyæ folder **Role** i klikn¹æ przycisk **Utwórz nowy obiekt** lub klikn¹æ folder **Role** prawym przyciskiem myszy, a nastêpnie wybraæ polecenia **Nowy** i **Rola**.
4. W wyœwietlonym oknie dialogowym wpisz nazwê nowej roli.
5. Kliknij przycisk **OK**.

Uwaga Zabezpieczenia pakietu nie zostan¹ w³¹czone, o ile do roli pakietu nie zostanie zamapowany prawid³owy u¿ytkownik.

Zobacz te¿

Mapowanie ról MTS do u¿ytkowników i grup, W³¹czanie zabezpieczenia pakietu MTS, Ustawianie to¿samoœci pakietu MTS, Folder "Role", Zarz¹dzanie u¿ytkownikami dla ról MTS

Mapowanie ról MTS do u¿ytkowników i grup

Podczas instalowania i rozmieszczania aplikacji, nale¿y zamapowaæ u¿ytkowników i grupy systemu Windows NT do jakichkolwiek istniej¹cych ról. Role okreœlaj¹ zakres dostêpu u¿ytkowników do sk³adników i interfejsów.

Aby przypisaæ u¿ytkowników do ról:

1. W lewym okienku programu MTS Explorer zaznacz pakiet zawieraj¹cy sk³adnik, do którego chcesz przypisaæ rolê.
2. Otwórz folder **Role**.
3. Kliknij dwukrotnie rolê, do której chcesz przypisaæ u¿ytkowników.
4. Otwórz folder **U¿ytkownicy**.
5. W menu **Akcja** kliknij polecenie **Nowy**. Mo¿esz równie zaznaczyæ folder **U¿ytkownicy** i klikn¹æ przycisk **Utwórz nowy obiekt** lub klikn¹æ folder **U¿ytkownicy** prawym przyciskiem myszy, a nastêpnie wybraæ polecenia **Nowy** i **U¿ytkownicy**.
6. W wyœwietlonym oknie dialogowym dodaj nazwy u¿ytkowników lub grup do roli. Aby odszukaæ interesuj¹ce Ciê konto u¿ytkownika, mo¿esz skorzystaæ z przycisków **Poka¿ u¿ytkowników** i **Szukaj**.
7. Kliknij przycisk **OK**.

Zobacz te¿

Dodawanie nowej roli MTS, W³¹czanie zabezpieczeñ pakietu MTS, Ustawianie to¿samoœci pakietu MTS, Folder "Role", Folder "U¿ytkownicy", Zarz¹dzanie u¿ytkownikami dla ról MTS

Utrzymanie w³aœciwego stanu pakietów MTS

Po pomyœlnym zbudowaniu, rozpowszechnieniu i rozmieszczeniu pakietu administratorzy mog¹ korzystaæ z programu MTS Explorer w celu utrzymania po¿¹danego stanu pakietów. Utrzymanie po¿¹danego stanu pakietów MTS wymaga monitorowania stanu i w³aœciwoœci samych pakietów i ich sk³adników oraz, jeœli to niezbêdne, ponownej konfiguracji ich w³aœciwoœci.

W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Korzystanie z arkuszy w³aœciwoœci w programie MTS Explorer

Zarz¹dzanie u¿ytkownikami dla ról MTS

Korzystanie z replikacji MTS

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Do monitorowania stanu i w³aœciwoœci pakietów aktywnych i ich sk³adników s³u¿¹ ustawienia **Stan** i **W³aœciwoœci** programu MTS Explorer. Aby obejrzeæ stan lub w³aœciwoœci wybranych elementów, nale¿y zaznaczyæ element, a nastêpnie klikn¹æ przycisk **Stan** lub **W³aœciwoœci** z paska narzêdzi programu MTS Explorer.

Przycisk **Stan** pozwala ogl¹daæ stan komputerów, pakietów i sk³adników. Na przyk³ad przycisk **Stan** u¿yty dla folderu "Komputery" pozwala sprawdziæ, czy zosta³a uruchomiona us³uga Microsoft Distributed Transaction Coordinator (MS DTC), a przycisk **Stan** u¿yty dla folderu "Pakiety zainstalowane" pozwala sprawdziæ, który pakiet uruchomiono. Przycisk **Stan** zastosowany do sk³adników pakietu pozwala obejrzeæ identyfikator programistyczny (progID) sk³adnika, obiekty aktywne oraz obiekty wywo³ane. Ponadto, przycisk **Stan** umo¿liwia szybkie sprawdzenie stanu zarz¹dzanych komputerów i procesów aplikacji.

Aby u¿yæ w programie MTS Explorer przycisku Stan

1. Zaznacz folder **Pakiety zainstalowane** lub **Sk³adniki**.
2. W prawym okienku programu MTS Explorer, na pasku narzêdzi MTS kliknij ikonê **Stan**.

Uwaga Przycisk **Stan** nie pozwala ogl¹daæ stanu pakietów narzêdziowych i systemowych.

Przycisk **W³aœciwoœci** s³u¿y do szybkiego przegl¹dania informacji o w³aœciwoœciach pakietów i sk³adników. Na przyk³ad przycisk u¿yty dla sk³adnika powoduje wyœwietlenie identyfikatora programistycznego sk³adnika (ProgID), ustawienia transakcji, lokalizacji biblioteki do³¹czanej dynamicznie (DLL), identyfikatora klasy (CLSID), modelu w¹tków oraz ustawieñ uwierzytelnienia zabezpieczeñ. Przycisk **W³aœciwoœci** mo¿e równie¿ pos³u¿yæ do szybkiego przegl¹du w³aœciwoœci transakcyjnych wszystkich sk³adników pakietu.

Aby u¿yæ w programie MTS Explorer przycisku W³aœciwoœci

1. W prawym okienku programu MTS Explorer zaznacz folder, którego stan chcesz obejrzeæ, na przyk³ad folder **Pakiety zainstalowane** lub **Sk³adniki.**
2. Na pasku narzêdzi programu MTS Explorer kliknij przycisk **W³aœciwoœci**. W³aœciwoœci elementów folderu zostan¹ wyœwietlone w prawym okienku programu MTS Explorer, w kolejnych kolumnach.

Zobacz te¿

Korzystanie z arkuszy w³aœciwoœci w programie MTS

Korzystanie z arkuszy w³aœciwoœci w programie MTS Explorer

Arkusze w³aœciwoœci pakietów i sk³adników pozwalaj¹ zmieniaæ konfiguracjê pakietów MTS. Na przyk³ad aby zmieniæ w³aœciwoœæ transakcji sk³adnika, mo¿na zmodyfikowaæ arkusz w³aœciwoœci dla ustawieñ transakcji. Przed zmian¹ konfiguracji ustawieñ konkretnej w³aœciwoœci warto zapoznaæ siê z tematami Tworzenie pakietów MTS i Instalacja pakietów MTS, które poœwiêcono opisowi procedur ustawiania ró¿nych w³aœciwoœci w programie MTS Explorer.

Pakiety mog¹ byæ blokowane, co ma za zadanie zapobiec próbom modyfikacji lub usuwania pakietów i ich sk³adników. Jeœli podczas instalacji lub rozmieszczania pakiet zostanie zablokowany, wówczas przed rozpoczêciem modyfikacji w³aœciwoœci sk³adnika, nale¿y go odblokowaæ. Wiêcej informacji na temat blokowania mo¿na znaleŸæ pod has³em Blokowanie pakietu MTS.

Aby skonfigurowaæ w³aœciwoœci aplikacji i pakietu za pomoc¹ stron w³aœciwoœci

1. Prawym przyciskiem myszy kliknij element, którego w³aœciwoœci chcesz ustawiæ. Mo¿esz równie¿ zaznaczyæ element w lewym okienku programu MTS Explorer i wybraæ polecenie **W³aœciwoœci** z menu **Akcja**.
2. Zmodyfikuj odpowiedni arkusz w³aœciwoœci i kliknij przycisk **OK** lub **Zastosuj**.
3. Odœwie¿ wszystkie sk³adniki pokazywane w oknie **Mój Komputer**, zaznaczaj¹c ikonê **Mój Komputer** i wybieraj¹c polecenie **Odœwie¿ wszystkie sk³adniki** z menu **Akcja**. Aby odœwie¿yæ pojedyncze pakiety, mo¿esz zaznaczyæ folder "Pakiety zainstalowane" i klikn¹æ ikonê **Odœwie¿**, wyœwietlan¹ w prawym okienku programu MTS Explorer na pasku narzedzi MTS.

   Uwaga Jeœli wczeœniej zmieniono poziom aktywacji pakietu, wówczas aby wprowadzone zmiany odnios³y skutek nale¿y zamkn¹æ proces pakietu. Proces pakietu mo¿na zamkn¹æ, klikaj¹c pakiet prawym przyciskiem myszy i wybieraj¹c opcjê **Zamknij**.

Zobacz te¿

Monitorowanie stanu i w³aœciwoœci w programie MTS Explorer

Zarz¹dzanie u¿ytkownikami dla ról MTS

Administrator mo¿e zarz¹dzaæ przypisanymi do ról u¿ytkownikami i grupami u¿ytkowników systemu Windows NT poprzez nastêpuj¹ce czynnoœci:

- Dodawanie nowego u¿ytkownika lub grupy do roli
- Usuwanie u¿ytkownika lub grupy z roli
- Przenoszenie u¿ytkownika lub grupy z jednej roli do innej

Wraz ze wzrostem liczby klientów okreœlonego pakietu serwera pojawia siê koniecznoœæ mapowania nowych u¿ytkowników do konkretnych ról pakietu. Wiêcej informacji na temat mapowania nowych u¿ytkowników do ról mo¿na znaleŸæ pod has³em Mapowanie ról MTS do u¿ytkowników i grup.

Czasami mo¿e wyst¹piæ koniecznoœæ usuniêcia u¿ytkowników z pewnej roli lub przeniesienia u¿ytkowników lub grup u¿ytkowników z istniej¹cej roli do roli nowo utworzonej. Na przyk³ad jeœli pracownik opuszcza firmê, nale¿y usun¹æ go ze wszystkich ról, z jakimi dotychczas by³ skojarzony jako pracownik firmy.

Kiedy w istniej¹cym pakiecie s¹ tworzone nowe role, mo¿e wyst¹piæ koniecznoœæ przeniesienia u¿ytkowników lub grup u¿ytkowników z dotychczasowych ról do nowych ról. Przypuœæmy, ¿e dla pakietu Ksiêgowoœæ utworzono rolê Urzêdnik i nale¿y przenieœæ czêœæ u¿ytkowników i grup u¿ytkowników z istniej¹cej roli Mened¿er (z uprawnieniami zapisu i odczytu danych z tabeli Ksiêgowoœæ) do roli Urzêdnik (wyposa¿onej tylko w uprawnienia odczytu). Aby przenieœæ u¿ytkowników lub grupy, nale¿y najpierw usun¹æ je z roli dotychczasowej, a nastêpnie zamapowaæ do nowej roli.

Aby usun¹æ u¿ytkownka z roli

1. Odszukaj pakiet zawieraj¹cy rolê, z której chcesz usun¹æ u¿ytkownika lub grupê. Zaznacz pakiet w lewym okienku programu MTS Explorer.
2. Otwórz folder **Role**.
3. Kliknij dwukrotnie rolê zdefiniowan¹ dla usuwanego u¿ytkownika.
4. Otwórz folder **U¿ytkownicy**.
5. Zaznacz u¿ytkownika, którego chcesz usun¹æ.
6. W menu **Akcja** kliknij polecenie **Usuñ**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy i wybraæ polecenie **Usuñ** z menu podrêcznego.
7. W wyœwietlonym oknie dialogowym kliknij przycisk **Tak**. U¿ytkownika mo¿esz usun¹æ tak¿e, zaznaczaj¹c reprezentuj¹c¹ go ikonê i naciskaj¹c klawisz **Delete**.

Zobacz te¿

Folder "Role", Folder "U¿ytkownicy", W³¹czanie zabezpieczeñ pakietu MTS, Ustawianie to¿samoœci pakietu MTS, Dodawanie nowej roli MTS, Mapowanie ról MTS do u¿ytkowników i grup

Korzystanie z replikacji MTS

Korzystanie z replikacji serwera MTS umo¿liwia program Microsoft Cluster Server (MSCS). Dziêki niemu, jeœli dany serwer w klastrze zacznie pracowaæ nieprawid³owo lub utraci zdolnoœæ komunikacji, wszystkie nieudane operacje mo¿e wykonaæ inny serwer.

W jaki sposób dokonaæ replikacji serwera MTS?

Aby dokonaæ replikacji serwera

1. Zainstaluj program MTS na obydwu komputerach. Wiêcej informacji znajdziesz pod has³em Konfigurowanie programu MTS za pomoc¹ programu Microsoft Cluster Server.
2. Wybierz komputer Ÿród³owy i zainstaluj pakiety w jego pamiêci.
3. Zarejestruj na obydwu komputerach wszystkie sk³adniki importowane.
4. SprawdŸ, czy na obydwu komputerach s¹ zainstalowane elementy dodatkowe, na przyk³ad biblioteki czasu wykonywania. (Zapoznaj siê z treœci¹ podrozdzia³u "Ograniczenia".)
5. W oknie **Mój Komputer** na karcie **Opcje** arkusza w³aœciwoœci w polu **Udzia³ replikacji** podaj nazwê komputera Ÿród³owego. (Zapoznaj siê z treœci¹ podrozdzia³u "Replikacja zawartoœci pakietu").
6. W wierszu poleceñ uruchom narzêdzie MTXREPL.EXE.

Narzêdzie wiersza poleceñ MTXREPL.EXE s³u¿y do replikacji serwera MTS. Obydwa komputery, Ÿród³owy i docelowy, musz¹ byæ uruchomione. W poleceniu MTXREPL.EXE nale¿y podaæ nastêpuj¹ce argumenty:

MTXREPL.EXE *obiekt*_Ÿród³owy *obiekt_docelowy*

Stosowana metoda replikacji zak³ada, ¿e istnieje jeden komputer Ÿród³owy, informacje o katalogach programu MTS s¹ kopiowane z pamiêci komputera Ÿród³owego do pamiêci komputera docelowego. Aby dokonaæ replikacji na inny komputer, nale¿y ponownie uruchomiæ narzêdzie MTXREPL.EXE. Zasada jednego komputera Ÿród³owego nie jest przestrzegana bezwzglêdnie   komputer docelowy mo¿e staæ siê komputerem Ÿród³owym dla kolejnej replikacji serwera MTS.

W trakcie replikacji program MTS ca³kowicie zastêpuje katalog komputera docelowego katalogiem komputera Ÿród³owego (wyj¹tki od tej zasady opisano w podrozdziale niniejszego tematu, "Czynnoœci nie wykonywane w trakcie replikacji MTS"). Wykonanie replikacji czêœciowej nie jest mo¿liwe. Jeœli replikacja siê nie powiedzie, mo¿na spróbowaæ ponownie uruchomiæ narzêdzie MTXREPL.EXE.

Za rozpoczêcie replikacji i sprawdzanie, czy zmiany wprowadzone na serwerze Ÿród³owym s¹ replikowane do wszystkich komputerów klastra odpowiadaj¹ administratorzy.

Zaleca siê, aby na komputerze docelowym replikacji nie by³y uruchomione ¿adne sk³adniki MTS. W trakcie uruchamiania programu MTS, podczas odczytu informacji z katalogu jest wyœwietlane okno awaryjne. Jeœli replikacja   zostanie wykonana w tym w³aœnie momencie, uruchomienie pakietu nie powiedzie siê i klient bêdzie musia³ podj¹æ ponown¹ próbê.

Replikacja pakietów IIS

Do replikacji pakietów IIS s³u¿y specjalne narzêdzie replikacji, program Microsoft Internet Information Server (IIS) w wersji 4.0. Replikacja wykonywana za pomoc¹ programu IIS polega na automatycznym kopiowaniu pe³nej informacji o katalogu MTS. Z tego powodu uruchamianie narzêdzia replikacji programu MTS nie jest konieczne; jeœli program IIS 4.0 nie jest zainstalowany, narzêdzie nie zostanie uruchomione.

Replikacja zawartoœci pakietu

W trakcie replikacji MTS s¹ kopiowane wszystkie niezbêdne biblioteki do³¹czane dynamicznie (DLL), biblioteki typów i biblioteki DLL typu proxy-stub sk³adników. Aby komputery docelowe replikacji mog³y uzyskaæ dostêp do tych plików, nale¿y wywo³aæ arkusz w³aœciwoœci okna **Mój Komputer**, zaznaczyæ kartê **Opcje** i w polu **Udzia³ replikacji** okreœliæ nazwê komputera Ÿród³owego replikacji. Na komputerze Ÿród³owym nale¿y utworzyæ ten udzia³, a nastêpnie udzieliæ komputerom docelowym uprawnienia odczytu (tylko odczytu). Nazwa okreœlona w polu **Udzia³ replikacji** odnosi siê do udzia³u, a nie do katalogu, z którym skojarzono nazwê udzia³u. Na przyk³ad jeœli katalog d:\mtx\repl jest wspó³u¿ytkowany jako MyReplPoint, w polu **Udzia³ replikacji** nale¿y podaæ "MyReplPoint", a nie "d:\mtx\repl".

Komputer okreœlony w polu **Udzia³ replikacji** musi udzieliæ pakietowi systemowemu komputera docelowego uprawnienie odczytu (tylko odczytu). Jeœli docelowy pakiet systemowy wykonuje sprawdzanie ról, wówczas to¿samoœæ pakietu systemowego na komputerze Ÿród³owym musi nale¿eæ do roli Administrator docelowego pakietu systemowego.

Lokalizacja biblioteki DLL pierwszego sk³adnika pakietu okreœla lokalizacjê w pamiêci komputera docelowego, gdzie zostan¹ zainstalowane pozosta³e pliki. Jeœli sk³adnik ten jest zapisany w podkatalogu domyœlnego katalogu pakietów, wówczas podkatalog taki jest tworzony równie¿ w pamiêci komputera docelowego. Domyœlny katalog pakietów jest okreœlany podczas instalacji programu MTS; na przyk³ad jeœli program MTS zosta³ zainstalowany w katalogu c:\program files\mtx, wówczas domyœlnym katalogiem instalacyjnym pakietu jest c:\program files\mtx\packages.

Jeœli replikowane pakiety s¹ zainstalowane wewn¹trz tego katalogu, wówczas w pamiêci komputera docelowego zostan¹ one zainstalowane w tej samej lokalizacji wzglêdnej, wewn¹trz domyœlnego katalogu instalacyjnego pakietów. Na przyk³ad:

Komputer	Domyœlny katalog pakietów	Katalog instalacyjny pakietów
□ród³owy	c:\program files\mtx\packages	c:\program files\mtx\packages\MyPak
Docelowy	d:\mts\packages	d:\mts\packages\MyPak

Pakiety zainstalowane na zewn¹trz domyœlnego katalogu instalacyjnego pakietów s¹ instalowane w podkatalogu "zewnêtrznym" komputera docelowego. Dla powy¿szego przyk³adu:

Komputer	Katalog instalacyjny pakietów
□ród³owy	Pakiet A w katalogu c:\program files\mtx\packages\MyPak Pakiet B w katalogu d:\misc
Docelowy	Pakiet A i pakiet B w katalogu d:\mts\packages\MyPak

Zaleca siê instalacjê pakietów wewn¹trz domyœlnego katalogu pakietów. Ponadto, wszystkie pliki danego pakietu powinny rezydowaæ w tym samym katalogu.

Administrowanie klastrami serwera programu MTS

Administruj¹c w klastrach MSCS, administratorzy musz¹ zawsze u¿ywaæ fizycznych nazw komputerów. Aby obejrzeæ kopiê komputera Ÿród³owego z klastra MSCS, nale¿y skorzystaæ z jego nazwy fizycznej. Aby przeprowadziæ replikacjê miêdzy komputerem Ÿród³owym i kopi¹, równie¿ nale¿y korzystaæ z fizycznych nazw komputerów.

Rozmieszczaj¹c aplikacje MTS w klastrze MSCS, nale¿y podaæ nazwê serwera wirtualnego. S³u¿y do tego pole **Nazwa serwera zdalnego**, znajduj¹ce siê na karcie **Opcje** arkusza w³aœciwoœci okna **Mój Komputer**. Jeœli pakiet jest eksportowany za pomoc¹ narzêdzia do obs³ugi plików wykonywalnych aplikacji, klienci instaluj¹cy aplikacjê zostan¹ skierowani do serwera wirtualnego. Jeœli w polu **Nazwa serwera zdalnego** nie zostanie okreœlona nazwa serwera, klienci zostan¹ skierowani do komputera fizycznego, co nie jest w³aœciw¹ konfiguracj¹ dla zabezpieczeñ w razie awarii.

Okreœlaj¹c ró¿ne nazwy serwerów wirtualnych mo¿na podzieliæ pakiety miêdzy ró¿ne komputery, zapewniaj¹c w ten sposób ochronê aplikacji przed awari¹. Okreœlaj¹c dwie nazwy serwerów wirtualnych, ka¿d¹ dla innego komputera fizycznego, mo¿na utworzyæ plik wykonywalny instalacji klienta dla pakietu zainstalowanego na obydwu komputerach. W ten sposób rozpowszechniana aplikacja mo¿e zostaæ rozdzielona miêdzy klientów.

Aby rozdzieliæ pakiety aplikacji MTS za pomoc¹ serwerów wirtualnych

1. Wybierz metodê podzia³u pakietów miêdzy komputery.

   Na przyk³ad mo¿esz uruchomiæ wszystkie pakiety jednoczeœnie na jednym komputerze, wszystkie pakiety na obydwu komputerach lub niektóre pakiety na pierwszym komputerze, a niektóre na drugim.
2. Za pomoc¹ programu MSCS Cluster Administrator utwórz nazwy serwerów wirtualnych dla obydwu komputerów.
3. Wyeksportuj pakiety, u¿ywaj¹c nazw serwerów wirtualnych utworzonych w kroku 2.

   Jeœli chcesz uruchomiæ wszystkie pakiety jednoczeœnie na pojedynczym komputerze, wyeksportuj je u¿ywaj¹c pojedynczej nazwy serwera wirtualnego. Jeœli chcesz uruchomiæ wszystkie pakiety jednoczeœnie na obydwu komputerach, wyeksportuj je podwójnie, osobno dla ka¿dej nazwy serwera wirtualnego. Do klientów musisz wówczas rozes³aæ dwa ró¿ne zestawy plików wykonywalnych instalacji klienta. Jeœli chcesz uruchomiæ po³owê pakietów na jednym komputerze, a po³owê na drugim, ka¿d¹ z nich wyeksportuj u¿ywaj¹c innej nazwy serwera wirtualnego.
4. Uruchom pliki wykonywalne klienta na komputerach klientów. W zale¿noœci od tego, jaka metoda podzia³u pakietów zosta³a wybrana w kroku 1, ka¿dy klient mo¿e wymagaæ uruchomienia ró¿nej liczby plików wykonywalnych.

Czynnoœci nie wykonywane w trakcie replikacji MTS

W trakcie replikacji MTS nie s¹ wykonywane nastêpuj¹ce czynnoœci:

- Instalacja programu MTS. Przed rozpoczêciem replikacji na wszystkich komputerach docelowych nale¿y zainstalowaæ program MTS.
- Zast¹pienie systemu MTS, pakietów narzêdziowych i pakietów narzêdziowych programu IIS. Pakiety te nale¿y skonfigurowaæ identycznie na obydwu komputerach, docelowym i Ÿród³owym (mimo ró¿nych œcie¿ek dostêpu dol plików). Program MTS nie wykonuje tego automatycznie; odpowiednia synchronizacja obydwu zestawów pakietów nale¿y do administratora.
- Replikacja informacji dotycz¹cych samego komputera, na przyk³ad w³aœciwoœci **Udzia³ replikacji** i **Nazwa serwera zdalnego**.
- Replikacja obiektów sk³adników programu MTS, które nie s¹ wykrywane przez program MTS, na przyk³ad bibliotek czasu wykonywania. Aby aplikacje dzia³a³y poprawnie, pliki te nale¿y skopiowaæ samodzielnie do ka¿dego komputera docelowego.
- Replikacja kont zabezpieczeñ komputerów lokalnych. W przypadku klastrów dla ról lub to¿samoœci pakietów korzystanie z takich kont nie jest zalecane.
- Replikacja danych trwa³ych, od których zale¿y poprawne dzia³anie aplikacji. Na przyk³ad Serwer SQL zapewnia automatyczn¹ ochronê swoich baz danych; nale¿y sprawdziæ, czy baza danych dla aplikacji jest skonfigurowana w sposób zapewniaj¹cy

ochronê przed awari¹.

Ograniczenia

- Replikacja MTS nie jest obs³ugiwana dla komputerów (Ÿród³owych lub docelowych) wyposa¿onych w system Windows 95.
- Replikacja MTS nie powiedzie siê, jeœli w komputerze Ÿród³owym nie istnieje ¿aden sk³adnik zdalny, który jest ustawiony do pracy na komputerze docelowym.
- Sk³adniki importowane mo¿na replikowaæ tylko wówczas, jeœli uprzednio zarejestrowano je w komputerach docelowych. Narzêdzie do replikacji automatycznie rejestruje i konfiguruje w komputerze-kopii sk³adniki zainstalowane. Replikuj¹c sk³adniki importowane z komputera Ÿród³owego, narzêdzie do replikacji oczekuje ich uprzedniego zarejestrowania w komputerze-kopii. Z tego wzglêdu u¿ytkownik powinien, jeszcze przed uruchomieniem narzêdzia do replikacji, zarejestrowaæ sk³adniki importowane rêcznie na wszystkich komputerach docelowych.
- W przypadku korzystania z programu MTS w wersji 2.0 lub wczeœniejszej, replikacja nie jest obs³ugiwana.

Zobacz te¿

Konfigurowanie programu MTS za pomoc¹ programu Microsoft Cluster Server, Opcje, karta (Komputer)

Zarz¹dzanie transakcjami MTS

Aby efektywnie zarz¹dzaæ transakcjami dla pakietów programu Microsoft Transaction Server (MTS), nale¿y dok³adnie zrozumieæ zasady dzia³ania rozpowszechnianych transakcji. W niniejszym podrozdziale opisano zasady dzia³ania transakcji oraz metody monitorowania i zarz¹dzania transakcjami za pomoc¹ programu MTS Explorer.

W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Podstawowe informacje o transakcjach MTS

Zarz¹dzanie us³ug¹ MS DTC

Monitorowanie transakcji MTS

Monitorowanie transakcji MTS w systemie Windows 95

Podstawowe informacje o stanach transakcji MTS

Przeprowadzanie transakcji MTS

Zobacz te¿

Lista transakcji, Statystyki transakcji, Ikony transakcji, Komunikaty œledzenia

Podstawowe informacje o transakcjach MTS

Program Microsoft Transaction Server (MTS) pozwala w ³atwy sposób u¿ywaæ, monitorowaæ i administrowaæ rozpowszechnianymi transakcjami we w³asnych aplikacjach. Rozpowszechniana transakcja jest to transakcja uaktualniona ze wzglêdu na zasoby chronione transakcjami w kilku systemach. Mened¿er transakcji, us³uga Microsoft Distributed Transaction Coordinator (MS DTC), u³atwia obs³ugê transakcji w systemach Windows NT i Windows 95. Us³uga MS DTC umo¿liwia równie¿ aktualizacjê dwóch lub wiêkszej liczby zasobów chronionych transakcjami w pojedynczym systemie.

Zobacz te¿

Zarz¹dzanie us³ug¹ MS DTC, Monitorowanie transakcji MTS, Monitorowanie transakcji MTS w systemie Windows 95, Podstawowe informacje o stanach transakcji MTS, Przeprowadzanie transakcji MTS

Zarz¹dzanie us³ug¹ MS DTC

Zarz¹dzanie transakcjami wymaga wczeœniejszego uruchomienia na serwerze MTS us³ugi Microsoft Distributed Transaction Coordinator (MS DTC). Jeœli us³uga MS DTC nie zostanie uruchomiona, klienci nie bêd¹ mogli uzyskaæ dostêpu do pakietów transakcyjnych.

Aby konfigurowaæ i uruchamiaæ klientów us³ugi MS DTC na komputerze wyposa¿onym w system Windows NT, u¿ytkownik musi mieæ dostêp do zapisu w rejestrze lokalnym oraz dostêp do odczytu zdalnego do klucza Software\Classes serwera.

Aby uruchomiæ lub zatrzymaæ us³ugê MS DTC:

1. W prawym okienku programu MTS Explorer zaznacz komputer, z którego zarz¹dzasz pakietami transakcyjnymi.
2. Otwórz menu **Akcja** i wybierz polecenie **Uruchom MS DTC** lub **Zatrzymaj MS DTC**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy i wybraæ z menu podrêcznego polecenie **Uruchom MS DTC** lub **Zatrzymaj MS DTC** .

Mo¿lliwa jest równie¿ konfiguracja ustawieñ DTC, na przyk³ad lokalizacji i rozmiarów pliku dziennika DTC. S³u¿y do tego karta **Zaawansowane**, dostêpna na arkuszach w³aœciwoœci **Komputer**.

Aby skonfigurowaæ ustawienia us³ugi MS DTC:

1. W prawym okienku programu MTS Explorer zaznacz komputer, z którego zarz¹dzasz pakietami transakcyjnymi.
2. Otwórz menu **Akcja** i wybierz polecenie **W³aœciwoœci**. Mo¿esz równie¿ klikn¹æ prawym przyciskiem myszy i wybraæ z menu podrêcznego polecenie **W³aœciwoœci** .
3. Zaznacz kartê **Zaawansowane**. Za pomoc¹ elementów karty mo¿esz zmieniæ sposób wyœwietlania transakcji w oknach **Lista transakcji** i **Komunikaty œledzenia**. Mo¿esz równie¿ zmieniæ lokalizacjê lub rozmiar pliku dziennika MS DTC. Zwiêkszenie rozmiaru dziennika pozwala uruchamiaæ wiêcej transakcji równoczeœnie.

Wa¿ne W programie MS DTC obowi¹zuje górna granica rozmiaru pliku dziennika:

- W systemie Windows NT maksymalny rozmiar dziennika wynosi 512 MB.
- W systemie Windows 95 maksymalny rozmiar dziennika wynosi 64 MB.

Nie mo¿na wykonywaæ kompresji pliku dziennika us³ugi MS DTC

Kompresja pliku dziennika MS DTC jest niemo¿liwa.

Usuwanie plików DTCXATM.LOG

Przed uaktualnieniem dotychczasowej instalacji do wersji MTS 2.0 nale¿y usun¹æ plik DTCXATM.LOG. Przed jego usuniêciem us³uga MS DTC musi zostaæ zatrzymana.

Zobacz te¿

Podstawowe informacje o transakcjach MTS, Monitorowanie transakcji MTS, Monitorowanie transakcji MTS w systemie Windows 95, Zaawansowane, karta (Komputer MTS), Lista transakcji, Statystyki transakcji, Ikony transakcji, Komunikaty œledzenia

Monitorowanie transakcji MTS

Administrowanie transakcjami w aplikacjach MTS umo¿liwiaj¹ specjalne okna transakcji programu MTS Explorer. W trzech oknach, **Komunikaty œledzenia**, **Statystyki transakcji** i **Lista transakcji**, s¹ wyœwietalne u¿yteczne informacje o stanie transakcji zarz¹dzanych przez us³ugê Microsoft Distributed Transaction Coordinator (MS DTC).

Okno **Lista transakcji** umo¿liwia obs³ugê stanów transakcji. Wiêcej informacji na ten temat mo¿na uzyskaæ pod has³em Przeprowadzanie transakcji.

Aby monitorowaæ transakcje za pomoc¹ okna Lista transakcji

1. W lewym okienku programu MTS Explorer zaznacz komputer zarz¹dzaj¹cy transakcj¹.
2. Kliknij dwukrotnie ikonê **Lista transakcji**.
3. Kliknij prawym przyciskiem myszy w prawym okienku programu, po czym wska¿ w menu podrêcznym polecenie **Widok**.
4. W podmenu **Widok** kliknij jedno z nastêpuj¹cych poleceñ:
   - **Du¿e ikony**

     Transakcje s¹ wyœwietlane w postaci du¿ych ikon.
   - **Ma³e ikony**

     Transakcje s¹ wyœwietlane w postaci ma³ych ikon.
   - **W³aœciwoœci**

     Transakcje s¹ wyœwietlane w pojedynczej kolumnie, obok której widoczna jest kolumna zawieraj¹ca skojarzone z transakcjami identyfikatory robocze (ang. unit-of-work ID). S¹ to unikatowe identyfikatory globalne transakcji, generowane przez us³ugê MS DTC w momencie rozpoczêcia transakcji. Na liœcie s¹ pokazane równie¿ transakcje podrzêdne i nadrzêdne.
   - **Lista**

     Transakcje s¹ wyœwietlane kolejno, w pojedynczej kolumnie.

Uwaga Polecenia **Ma³e ikony** i **Lista** pozwalaj¹ wyœwietliæ najwiêcej transakcji jednoczeœnie. Polecenie **W³aœciwoœci** powoduje wyœwietlenie najwiêkszej liczby informacji o transakcjach, a polecenie **Du¿e ikony** pozwala przedstawiæ transakcje w formie najbardziej czytelnej.

Aby obejrzeæ statystyki transakcji

1. W lewym okienku programu MTS Explorer zaznacz komputer, na którym chcesz obejrzeæ statystyki transakcji.
2. Kliknij dwukrotnie ikonê **Statystyki transakcji**.

Aby obejrzeæ komunikaty œledzenia

1. W lewym okienku programu MTS Explorer zaznacz komputer zarz¹dzaj¹cy transakcj¹.
2. Kliknij dwukrotnie ikonê **Komunikaty œledzenia**.

Okno **Statystyki transakcji MS DTC** jest wyœwietlane w lewym okienku programu.

W³aœciwoœci transakcji

W³aœciwoœci transakcji mo¿na obejrzeæ, klikaj¹c wybran¹ transakcjê prawym przyciskiem myszy, a nastêpnie wybieraj¹c z menu podrêcznego polecenie **W³aœciwoœci**. Polecenie **W³aœciwoœci** powoduje wyœwietlenie listy wszystkich mened¿erów transakcji, którzy uczestnicz¹ w transakcji. W przypadku transakcji podrzêdnych polecenie powoduje wyœwietlenie listy tymczasowych transakcji nadrzêdnych. W przypadku transakcji nadrzêdnych s¹ wyœwietlane tymczasowe transakcje podrzêdne.

Zobacz te¿

Podstawowe informacje o transakcjach MTS, Monitorowanie transakcji MTS w systemie Windows 95, Podstawowe informacje o stanach transakcji MTS, Przeprowadzanie transakcji MTS, Lista transakcji, Statystyki transakcji, Ikony transakcji, Komunikaty œledzenia

Monitorowanie transakcji MTS w systemie Windows 95

W systemach Windows 95 i Windows NT do zarz¹dzania transakcjami MTS s³u¿¹ nastêpuj¹ce okna programu MTS Explorer:

- **Statystyki transakcji**
- **Lista transakcji**
- **Komunikaty œledzenia**

Domyœlnie, us³uga Microsoft Distributed Transaction Coordinator (MS DTC) jest skonfigurowana tak, aby by³a uruchamiana automatycznie po uruchomieniu systemu Windows NT lub Windows 95. Aby zapobiec automatycznemu uruchamianiu us³ugi MS DTC po ponownym rozruchu komputera wyposa¿onego w system Windows 95, nale¿y skorzystaæ z Edytora rejestru, aby odszukaæ klucz rejestru HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices i usun¹æ wartoœæ wpisu z rejestru o nazwie MSDTC. Aby ponownie w³¹czyæ automatyczne uruchamianie us³ugi MS DTC, nale¿y utworzyæ (za pomoc¹ Edytora rejestru) wpis do rejestru o nazwie MSDTC i nadaæ mu wartoœæ msdtc-start. Wpis znajduje siê w kluczu rejestru HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunService .

Uwaga Aby konfigurowaæ i uruchamiaæ klientów us³ugi MS DTC na komputerze wyposa¿onym w system Windows NT, u¿ytkownik musi mieæ dostêp do zapisu w rejestrze lokalnym oraz dostêp do odczytu zdalnego klucza Software\Classes serwera.

Aby administrowaæ zdalnie komputerem wyposa¿onym w system Windows NT za pomoc¹ komputera wyposa¿onego w system Windows 95, nale¿y zainstalowaæ us³ugê Rejestr zdalny dla systemu Windows 95. Us³uga ta pozwala zmieniaæ wpisy do rejestru dla zdalnego komputera z systemami Windows NT (dysponuj¹c oczywiœcie odpowiednimi uprawnieniami). Aby zainstalowaæ us³ugê Rejestr zdalny, nale¿y u¿yæ dysku CD-ROM systemu Windows 95 i przejœæ do podkatalogu \Admin\Nettols\Remotereg. Opis procedury instalacji us³ugi znajduje siê w pliku Regserv.txt.

Poniewa¿ w systemie Windows 95 nie istnieje dziennik zdarzeñ, zdarzenia us³ugi MS DTC s¹ przechowywane w pliku tekstowym o nazwie msdtc.txt. Plik ten znajduje siê w podkatalogu \MTSLogs katalogu Windows i zawiera informacje o zdarzeniach us³ugi MS DTC.

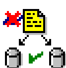
Zobacz te¿

Podstawowe informacje o transakcjach MTS, Monitorowanie transakcji MTS, Lista transakcji MTS, Statystyki transakcji, Ikony transakcji, Komunikaty œledzenia

Podstawowe informacje o stanach transakcji MTS

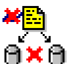
Poprawne zarz¹dzanie transakcjami wymaga zrozumienia znaczenia poszczególnych stanów transakcji i ich wp³ywu na administrowany pakiet MTS. Stany transakcji mo¿na obejrzeæ w oknie **Lista transakcji** programu Microsoft Distributed Transaction Coordinator (MS DTC), a reprezentuj¹ je nastêpuj¹ce ikony (widoczne w widoku "Du¿e ikony"):

Ikona	Opis
	Aktywna Transakcja zosta³a uruchomiona.
	Przerywana Transakcja ma byæ przerwana. Us³uga MS DTC powiadamia wszystkich uczestników, ¿e transakcja musi zostaæ przerwana. W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.
	Przerwana Transakcja zosta³a przerwana. Wszyscy uczestnicy zostali powiadomieni. Po przerwaniu transakcja jest natychmiast usuwana z listy transakcji wyœwietalnej w oknie **Transakcje MS DTC**. W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.
	W przygotowaniu Aplikacja klienta wys³a³a ¿¹danie przekazania transakcji. Us³uga MS DTC zbiera informacje o przygotowaniach od wszystkich uczestników.
	Przygotowana Wszyscy uczestnicy odpowiedzieli twierdz¹co na pytanie o przygotowanie transakcji.
	W¹tpliwa Transakcja jest przygotowana, koordynuje j¹ inna us³uga MS DTC, a koordynuj¹ca us³uga MS DTC jest niedostêpna. Administrator systemu mo¿e wymusiæ przekazanie lub przerwanie transakcji, klikaj¹c w oknie **Transakcje** prawym przyciskiem myszy, a nastêpnie wybieraj¹c polecenia **PrzeprowadŸ/Przeka¿** lub **PrzeprowadŸ/Przerwij**. Po wymuszeniu wyniku transakcja jest traktowana jako "przekazana w sposób wymuszony" lub "przerwana w sposób wymuszony". **Uwaga** Bez znajomoœci tematu Przeprowadzanie transakcj MTSi nie nale¿y wymuszaæ wyniku transakcji w¹tpliwej.



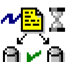
Wymuszone przekazanie

Administrator wymusi³ przekazanie transakcji w¹tpliwej (Warto zapoznaæ siê z tematem Przeprowadzanie transakcji MTS).



Wymuszone przerwanie

Administrator wymusi³ przerwanie transakcji w¹tpliwej (Warto zapoznaæ siê z tematem Przeprowadzanie transakcji).



Przekazywana

Transakcja zosta³a przygotowana pomyœlnie i us³uga MS DTC powiadamia uczestników, ¿e transakcja zosta³ przekazana. Us³uga MS DTC nie mo¿e zakoñczyæ transakcji, dopóki wszyscy uczestnicy nie potwierdz¹ odbioru (i zapisz¹ w dzienniku) ¿¹dania przekazania.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

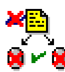
Nie mo¿na powiadomiæ o przerwaniu

Us³uga MS DTC powiadomi³a wszystkich pod³¹czonych uczestników, ¿e transakcja zosta³a przerwana. Nie zostali powiadomieni jedynie uczestnicy niedostêpni.

Ten stan transakcji wystêpuje wówczas, kiedy us³uga musi powiadomiæ mened¿era zasobów (na przyk³ad system IBM LU 6.2) o przerwaniu transakcji, ale nie mo¿e tego zrobiæ, poniewa¿ po³¹czenie z systemem firmy IBM zosta³o przerwane.

Administrator systemu mo¿e wymusiæ na us³udze zapomnienie transakcji, klikaj¹c w oknie **Transakcje** prawym przyciskiem myszy, a nastêpnie wybieraj¹c polecenie **PrzeprowadŸ/Zapomnij**.

Uwaga Bez znajomoœci tematu Przeprowadzanie transakcji proszê nie wymuszaæ rêcznie zapominania transakcji.

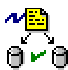


Nie mo¿na powiadomiæ o przekazaniu

Us³uga MS DTC powiadomi³a wszystkich pod³¹czonych uczestników, ¿e transakcja zosta³a przekazana. Nie zostali powiadomieni jedynie uczestnicy niedostêpni.

Administrator systemu mo¿e wymusiæ na us³udze zapomnienie transakcji, klikaj¹c prawym przyciskiem myszy w oknie **Transakcje**, a nastêpnie wybieraj¹c polecenie **PrzeprowadŸ/Zapomnij**.

Uwaga Bez znajomoœci tematu Przeprowadzanie transakcji proszê nie wymuszaæ rêcznie zapominania transakcji.



Przekazana

Transakcja zosta³a przekazana, a wszyscy uczestnicy zostali o tym powiadomieni. Po przekazaniu transakcja jest natychmiast usuwana z listy transakcji wyœwietalnej w oknie **Transakcje MS DTC**.

W tym momencie nie istnieje mo¿liwoœæ zmiany wyniku transakcji.

Zobacz te¿

Podstawowe informacje o transakcjach MTS, Monitorowanie transakcji MTS, Monitorowanie transakcji MTS w systemie Windows 95, Przeprowadzanie transakcji MTS, Lista transakcji

Przeprowadzanie transakcji MTS

Administrator transakcji MTS czasami staje przed koniecznoœci¹ rêcznego przeprowadzenia transakcji dla pewnej aplikacji MTS. Do przeprowadzania transakcji s³u¿y okno **Lista transakcji** programu MTS Explorer. W oknie tym mo¿na wybraæ jedno z nastêpuj¹cych poleceñ:

- **Przeka¿** Polecenie to wymusza przekazanie transakcji.
- **Przerwij** Polecenie to wymusza przerwanie transakcji i przywrócenie jej stanu pierwotnego.
- **Zapomnij** Polecenie to powoduje usuniêcie przekazanej lub przerwanej transakcji z dziennika us³ugi Microsoft Distributed Transaction Coordinator (MS DTC). Przed wykonaniem tego polecenia nale¿y zawsze wymusiæ wynik transakcji.

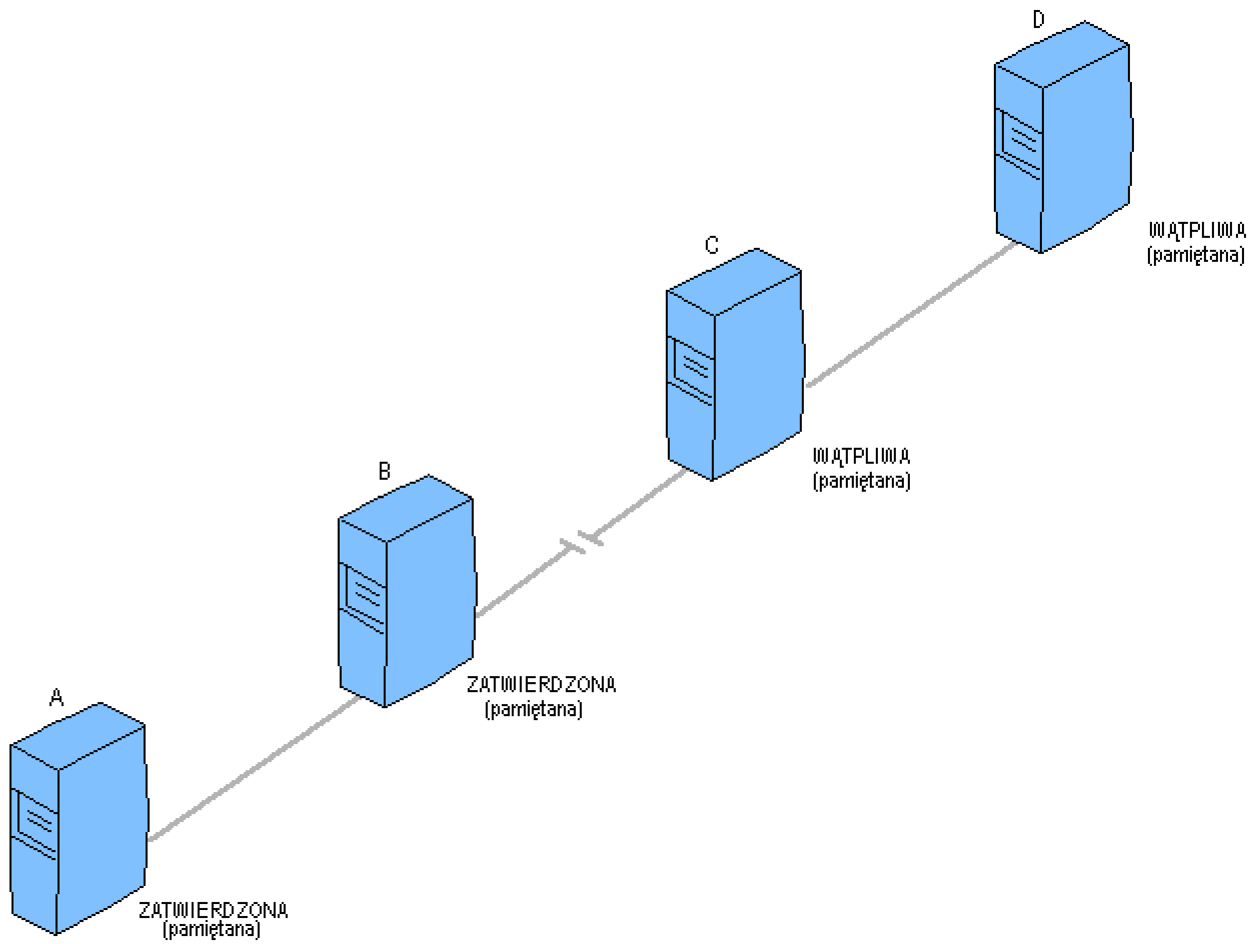
Czasami zachodzi koniecznoœæ przekazania lub przerwania transakcji w celu zwolnienia blokad, a tym samym udostêpnienia zasobów bazy danych innym aplikacjom i u¿ytkownikom sieci.

Sytuacja taka mo¿e wyst¹piæ na przyk³ad w wyniku przerwania linii komunikacyjnej miêdzy dwoma komputerami w sieci. Po rêcznym przekazaniu lub przerwaniu transakcji czêsto nale¿y równie¿ wymusiæ "zapomnienie" transakcji, co oznacza jej usuniêcie z lokalnego pliku dziennika us³ugi MS DTC.

Rozwa¿my przyk³ad rêcznego przerwania transakcji. Za³ó¿my, ¿e spe³nione zosta³y nastêpuj¹ce warunki:

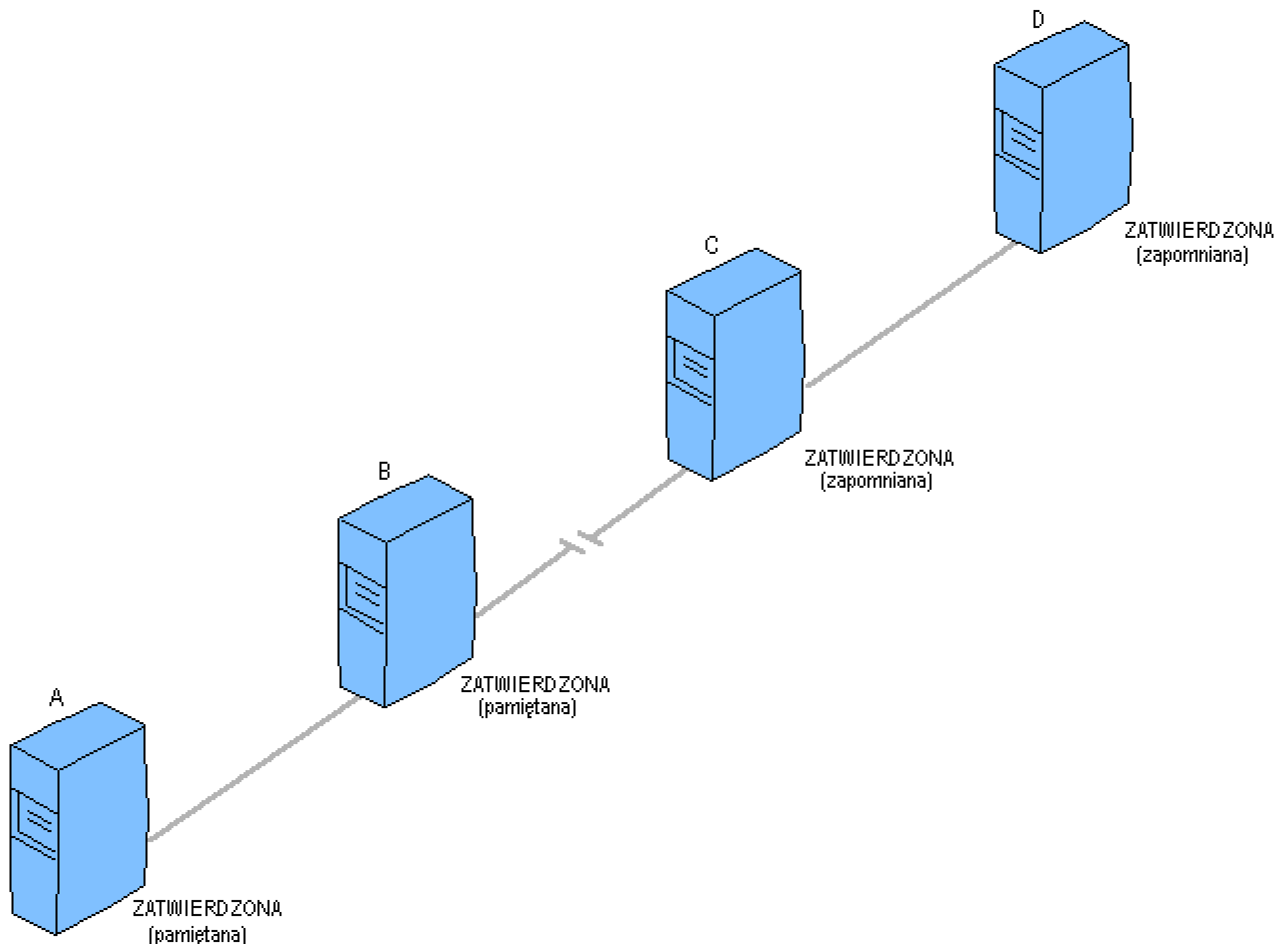
- Przekaz koordynuje us³uga MS DTC uruchomiona na komputerze A.
- Komunikacja miêdzy komuterem A i D jest prowadzona sekwencyjnie, na podstawie protoko³u przekazu dwufazowego.
- Pierwsza czêœæ protoko³u przekazu dwufazowego zosta³a zakoñczona i us³uga MS DTC zapisa³a rekord COMMITTED do swojego dziennika.
- W trakcie drugiej czêœci protoko³u przekazu dwufazowego komunikacja miêdzy komputerami B i C zosta³a przerwana.

Transakcja zosta³a przerwana w nastêpuj¹cym stanie:

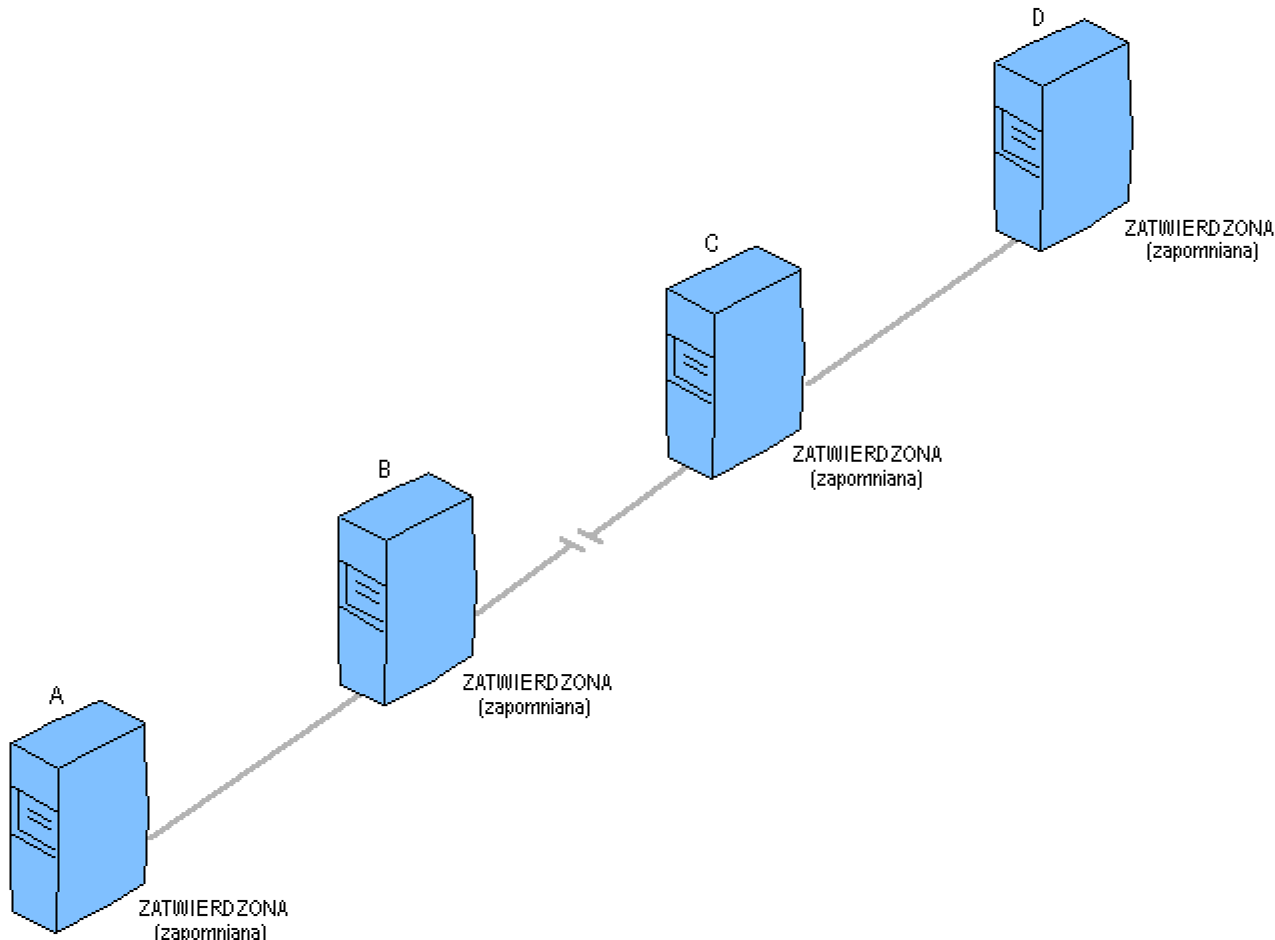


Poniewa¿ linia komunikacyjna miêdzy komputerami A i B jest nadal nienaruszona, komputer B przekaza³ transakcjê. Niemniej jednak, obydwa komputery musz¹ zachowaæ w swoich plikach dzienników rekordy COMMITTED, a¿ do momentu kiedy komputery C i D potwierdz¹, ¿e równie¿ przekaza³y transakcjê. Aby przeprowadziæ transakcjê (a tym samym zwolniæ blokady baz danych na³o¿one na komputery C i D), administrator systemu wymusza przekazanie transakcji przez komputer C (zobacz nastêpny rysunek).

Poniewa¿ linia komunikacyjna miêdzy komputerami A i B jest nadal nienaruszona, przekazanie transakcji wymuszone na komputerze C pozwala przekazaæ transakcjê komputerowi D. Na komputerze D mo¿e zostaæ zwolniona blokada bazy danych, a transakcja mo¿e zostaæ zapomniana. Kiedy komputer D wyœle do komputera C potwierdzenie o przekazaniu i zapomnieniu transakcji, na komputerze C równie¿ bêdzie mo¿na zwolniæ blokady bazy danych i zapomnieæ transakcjê.



Transakcjê przekazano do wszystkich komputerów. Tym niemniej, poniewa¿ komputer C nie mo¿e poinformowaæ o przekazaniu relacji komputera B, komputer B nadal musi pamiêtaæ transakcjê. Poniewa¿ na komputerze B transakcja nie zosta³a zapomniana, komputer A równie¿ musi j¹ pamiêtaæ. Aby zakoñczyæ transakcjê, administrator systemu zmusza komputer B do zapomnienia transakcji (zobacz nastêpny rysunek). To z kolei pozwala zapomnieæ transakcjê na komputerze A. W ten sposób protokó³ dwufazowego przekazu zosta³ wykonany rêcznie, a transakcja zakoñczona.



Wa¿ne Z powodu dwustronnoœci komunikacji opartej na protokole dwufazowego przekazu zaleca siê przeprowadzaæ transakcje rêcznie, jeœli tylko komputery nastêpuj¹ce po sobie przerw¹ komunikacjê. W opisanym przyk³adzie przekaz nale¿a³o wymusiæ na komputerze C (nie D), a zapomnienie transakcji na komputerze B (nie A).

Jeœli po awarii systemu lub po³¹czenia systemy uczestnicz¹ce w transakcji zostan¹ uruchomione ponownie, a po³¹czenia przywrócone, us³uga MS DTC przeprowadza transakcje automatycznie. Us³uga nie jest w stanie przeprowadziæ transakcji, jeœli systemy uleg³y awarii, a po³¹czenia nie zosta³y przywrócone. W takim wypadku nale¿y rêcznie przeprowadziæ transakcje, które znajduj¹ siê w stanach: "W¹tpliwa", "Nie mo¿na powiadomiæ o przerwaniu", "Nie mo¿na powiadomiæ o przekazaniu".

Stan "W¹tpliwa"

Stan *w¹tpliwa* odnosi siê do sytuacji, kiedy transakcja jest transakcj¹ podrzêdn¹, us³uga MS DTC jest przygotowana, a nadrzêdna us³uga MS DTC jest niedostêpna. Aby przeprowadziæ transakcjê w¹tpliw¹, wykonaj nastêpuj¹ce kroki:

1. W oknie **Lista transakcji** odszukaj bezpoœredni¹ transakcjê nadrzêdn¹ transakcji w¹tpliwej. W tym celu kliknij prawym przyciskiem myszy i wybierz polecenie **W³aœciwoœci**. W jego efekcie zostan¹ wyœwietlone komputery nadrzêdnej us³ugi MS DTC i podrzêdnej us³ugi MS DTC, przygotowane do transakcji.
2. Odszukaj nadrzêdn¹ us³ugê MS DTC, a nastêpnie w oknie **Lista transakcji** wyznacz wynik transakcji w¹tpliwej.

- Jeœli transakcja nie jest wyœwietlana w oknie **Lista transakcji**, oznacza to, ¿e transakcja zosta³a przerwana i mo¿esz przerwaæ j¹ rêcznie na komputerze podrzêdnym.
- Jeœli transakcja jest oznaczona na komputerze podrzêdnym jako "Nie mo¿na powiadomiæ o przekazaniu", oznacza to, ¿e transakcja zosta³a przekazana i mo¿esz przekazaæ j¹ rêcznie na komputerze podrzêdnym.
- Jeœli transakcja jest oznaczona na komputerze podrzêdnym jako "Nie mo¿na powiadomiæ o przerwaniu", oznacza to, ¿e transakcja zosta³a przerwana i mo¿esz przerwaæ j¹ rêcznie na komputerze podrzêdnym.
- Jeœli transakcja jest oznaczona na komputerze nadrzêdnym jako "W¹tpliwa", musisz, w oknie **Lista transakcji** na komputerze nadrzêdnym, odszukaæ nastêpny bezpoœredni komputer nadrzêdny. Kontynuuj przeszukiwanie drzewa przekazywania, a¿ do momentu znalezienia komputera nadrzêdnego, w którym transakcja jest albo nie wyœwietlana (co oznacza, ¿e jest przerwana), albo znajduje siê w stanie "Nie mo¿na powiadomiæ o przerwaniu" (co oznacza, ¿e jest przerwana), albo znajduje siê w stanie "Nie mo¿na powiadomiæ o przekazaniu" (co oznacza, ¿e jest przekazana). Jeœli na komputerze nadrzêdnym transakcja jest przerwana, rêcznie wymuœ jej przerwanie w bezpoœrednim komputerze podrzêdnym. Jeœli na komputerze nadrzêdnym transakcja jest przekazana, rêcznie wymuœ jej przekazanie w bezpoœrednim komputerze podrzêdnym.

3. Po rêcznym przerwaniu lub przekazaniu transakcji na komputerze podrzêdnym rêcznie wymuœ jej zapomnienie w bezpoœrednim komputerze nadrzêdnym.

Stan "Nie mo¿na powiadomiæ o przekazaniu"

Stan "Nie mo¿na powiadomiæ o przekazaniu" oznacza, ¿e transakcja zosta³a przekazana, ale niektóre podrzêdne us³ugi MS DTC nie zosta³y o tym powiadomione. W takim wypadku nale¿y przeprowadziæ transakcjê rêcznie. Po pierwsze, nale¿y klikn¹æ transakcjê prawym przyciskiem myszy. Zostan¹ wówczas wyœwietlone nadrzêdne i podrzêdne us³ugi MS DTC transakcji. Po zlokalizowaniu podrzêdnych us³ug MS DTC nale¿y w ka¿dej z nich wymusiæ przekazanie transakcji. Na sam koniec nale¿y powróciæ do us³ugi MS DTC, która pokazywa³a transakcjê w stanie "Nie mo¿na powiadomiæ o przekazaniu" i wymusiæ zapomnienie transakcji.

Uwaga Nie nale¿y wymuszaæ rêcznie zapominania transakcji, dopóki o wyniku transakcji nie zostan¹ powiadomione wszystkie podrzêdne us³ugi MS DTC.

Stan "Nie mo¿na powiadomiæ o przerwaniu"

Stan "Nie mo¿na powiadomiæ o przerwaniu" oznacza, ¿e transakcja zosta³a przerwana, ale niektóre podrzêdne us³ugi MS DTC nie zosta³y o tym powiadomione. Stan ten jest to¿samy ze stanem "Przerywana". Aby przeprowadziæ transakcjê rêcznie, nale¿y wykonaæ nastêpuj¹ce kroki. Po pierwsze, nale¿y klikn¹æ transakcjê prawym przyciskiem myszy. Zostan¹ wówczas wyœwietlone nadrzêdne i podrzêdne us³ugi MS DTC transakcji. Po zlokalizowaniu podrzêdnych us³ug MS DTC nale¿y w ka¿dej z nich wymusiæ przerwanie transakcji. Na sam koniec nale¿y powróciæ do us³ugi MS DTC, która pokazywa³a transakcjê w stanie "Nie mo¿na powiadomiæ o przerwaniu" i wymusiæ zapomnienie transakcji.

Uwaga Nie nale¿y wymuszaæ rêcznie zapominania transakcji, dopóki o wyniku transakcji nie zostan¹ powiadomione wszystkie podrzêdne us³ugi MS DTC.

Aby przeprowaadziæ transakcjê

1. W lewym okienku programu MTS Explorer zaznacz komputer, na którym chcesz przeprowadziæ transakcjê.

2 Kliknij dwukrotnie ikonê **Lista transakcji**.

3 W prawym okienku kliknij prawym przyciskiem myszy nad transakcj¹, któr¹ chcesz przeprowadziæ.

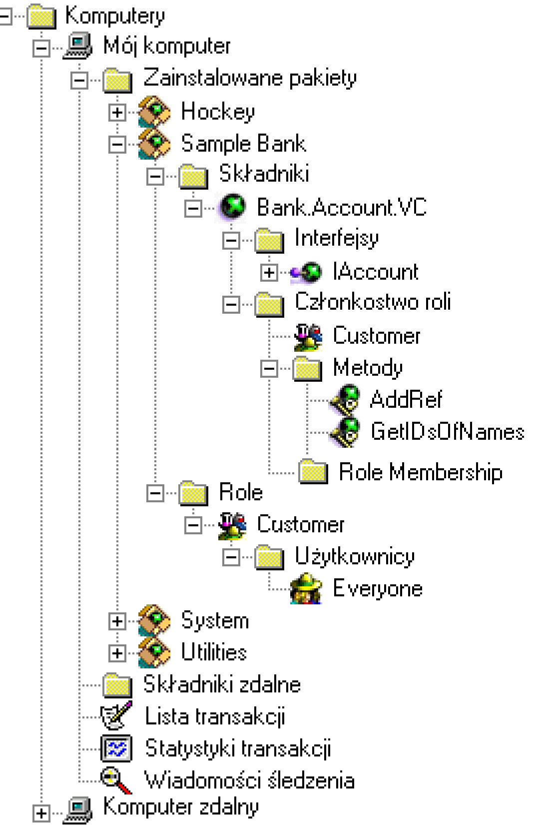
4 W podmenu **PrzeprowadŸ** kliknij polecenie **Przeka¿**, **Przerwij** lub **Zapomnij**.

Zobacz te¿

Podstawowe informacje o transakcjach MTS, Zarz¹dzanie us³ug¹ MS DTC, Monitorowanie transakcji MTS, Monitorowanie transakcji MTS w systemie Windows 95, Podstawowe informacje o stanach transakcji MTS, Lista transakcji

Automatyzacja czynnoœci administracyjnych w programie MTS

Projektanci i zaawansowani administratorzy sieci Web, korzystaj¹cy z programu Microsoft Transaction Server (MTS), maj¹ do dyspozycji szereg skryptowych obiektów administracyjnych pozwalaj¹cych automatyzowaæ administrowanie i rozmieszczanie aplikacj MTS. Obiekty skryptowe odpowiadaj¹ hierarchii kolekcji programu Microsoft Transaction Server Explorer. Wywo³uj¹c interfejsy odpowiednich obiektów skryptowych mo¿na automatyzowaæ wiele procedur administracyjnych. W poni¿szym diagramie ukazano rodzaje kolekcji administrowanych i rozpowszechnianych za pomoc¹ programu MTS Explorer:



Efektywne korzystanie ze skryptowych obiektów administracyjnych wymaga dok³adnej wiedzy o zadaniach automatyzowanych w programie MTS Explorer. Poniewa¿ skryptowe obiekty administracyjne s¹ pobierane z interfejsu **Idispatch**, projektuj¹c aplikacje mo¿na korzystaæ z dowolnego jêzyka programowania obs³uguj¹cego Automatyzacjê OLE. Zalecanymi narzêdziami projektowymi s¹: program Microsoft Visual Basic™ w wersji 5.0 oraz program Microsoft Visual C++ w wersji 5.0. Obydwa obs³uguj¹ technologiê ActiveX. Wyczerpuj¹ce informacje na temat korzystania ze skryptowych obiektów administracyjnych oraz przyk³ad kodu jêzyka Visual Basic pokazuj¹cy ich zastosowania mo¿na znaleŸæ w ksi¹¿ce *MTS Administrative Reference*.

W niniejszym podrozdziale omówiono nastêpuj¹ce tematy:

Obiekty administracyjne programu MTS

Przyk³ady skryptów napisanych w jêzyku Visual Basic, automatyzuj¹cych administrowanie

w programie MTS

Przyk³adowa aplikacja utworzona za pomoc¹ programu Visual Basic, automatyzuj¹ca administrowanie w programie MTS

Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Zobacz te¿

Podrêcznik MTS Administrative Reference

Obiekty administracyjne programu MTS

Do administrowania s³u¿¹ nastêpuj¹ce obiekty skryptowe:

Obiekt Catalog

Obiekt CatalogObject

Obiekt CatalogCollection

Obiekt PackageUtil

Obiekt ComponentUtil

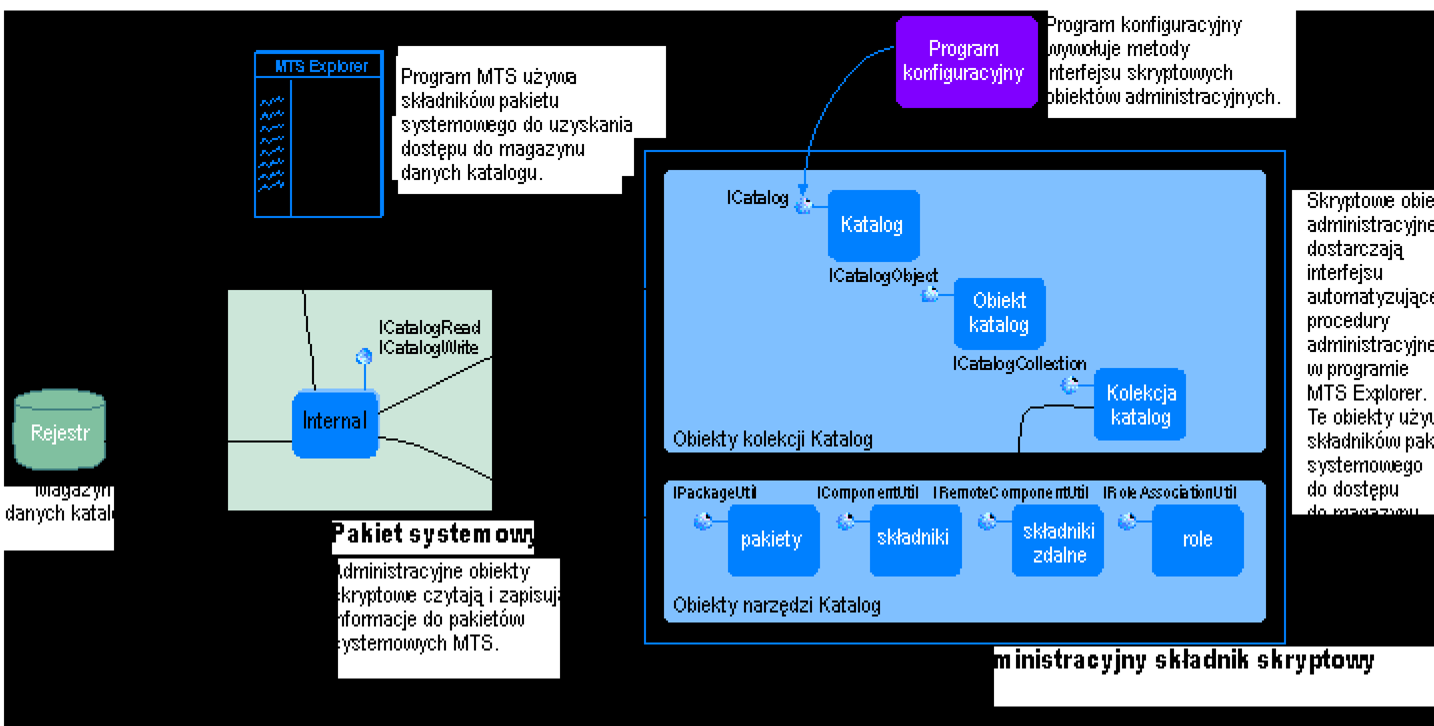
Obiekt RemoteComponentUtil

Obiekt RoleAssociationUtil

Powy¿sze obiekty dostarczaj¹ interfejsów ogólnych i narzêdziowych, u¿ytecznych w tworzeniu ró¿nego rodzaju skryptów. Obiekty Catalog, CatalogObject i CatalogCollection s¹ zwi¹zane z warstw¹ obiektów kolekcji katalogów i pozwalaj¹ kierowaæ takimi czynnoœciami najwy¿szego poziomu, jak tworzenie i modyfikowanie obiektów. Obiekt Catalog pozwala ³¹czyæ siê z konkretnymi serwerami i uzyskiwaæ dostêp do kolekcji. Wywo³uj¹c obiekt CatalogCollection mo¿na wyliczaæ, tworzyæ, usuwaæ i modyfikowaæ obiekty, jak równie¿ uzyskiwaæ dostêp do zwi¹zanych z nimi kolekcji. Obiekt CatalogObject s³u¿y do pobierania i ustawiania w³aœciwoœci obiektów.

Obiekty nale¿¹ce do warstwy obiektów narzêdzi katalogowych   PackageUtil, ComponentUtil, RemoteComponentUtil i RoleAssociationUtil   pozwalaj¹ automatyzowaæ zadania bardziej szczegó³owe, takie jak instalacja sk³adników i eksport pakietów. Powy¿sza warstwa narzêdziowa daje równie¿ mo¿liwoœæ programowania zadañ zwi¹zanych z typami kolekcji, na przyk³ad przypisywania u¿ytkownikom lub klasom u¿ytkowników ról.

Poni¿szy diagram pokazuje, w jaki sposób przyk³adowy program s³u¿¹cy do konfiguracji wykorzystuje metody oparte na obiektach administracyjnych w celu odczytu i zapisu danych w katalogu.



Wiêcej informacji na temat uzyskiwania przyk³adów programów administracyjnych i poœwiêconej im dokumentacji mo¿na znaleŸæ pod has³em Instalacja przyk³adowych projektów i dokumentacji programu MTS.

Zobacz te¿

MTS Administrative Reference

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Korzystanie z obiektów kolekcji katalogu programu MTS

W programie MTS Explorer ka¿dy folder odpowiada kolekcji przechowywanej w katalogu. Obiekty Catalog, CatalogObject i CatalogCollection nale¿¹ do warstwy obiektów kolekcji katalogu. Obiekty te pozwalaj¹ automatyzowaæ w aplikacjach ogólne procedury administracyjne, na przyk³ad operacje tworzenia, usuwania i modyfikowania obiektów.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Tworzenie instancji obiektu CatalogCollection

Instancje obiektu CatalogCollection daj¹ mo¿liwoœæ uzyskiwania dostêpu do dowolnych typów kolekcji lub danych, zapisanych w dowolnym folderze programu MTS Explorer. Aby utworzyæ instancjê obiektu CatalogCollection, nale¿y u¿yæ metody **GetCollection** i okreœliæ nazwê kolekcji. Obiekt CatalogCollection zapewnia dostêp do kolekcji dowolnego typu (na przyk³ad danych z jakiegokolwiek folderu programu MTS Explorer). Wywo³anie metody **GetCollection** z obiektu Catalog powoduje uzyskanie dostêpu do kolekcji najwy¿szego poziomu. Wywo³anie metody **GetCollection** z obiektu CatalogCollection zapewnia dostêp do odpowiedniej kolekcji ni¿szego poziomu.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Wype³nianie obiektu CatalogCollection danymi

Aby obiekt CatalogCollection wype³niæ danymi z katalogu, nale¿y u¿yæ obiektów administracyjnych. Warto zauwa¿yæ, ¿e tworzenie instancji nie oznacza odczytywania z katalogu ¿adnych danych. Aby móc przegl¹daæ lub zmieniaæ dane kolekcji, nale¿y wywo³aæ metody **Populate** lub **PopulateByKey**. Metoda **Populate** pozwala odczytaæ do kolekcji wszystkie dane. Metoda **PopulateByKey** pozwala odczytaæ tylko dane wyszczególnione. Dodawanie danych do kolekcji oraz korzystanie z interfejsów obiektów narzêdziowych nie wymaga wype³niania kolekcji danymi.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Pobieranie obiektów programu MTS i pobieranie/ustawianie w³aœciwoœci

Obiekt CatalogCollection zawiera metody pozwalaj¹ce iterowaæ obiekty kolekcji (wywo³ywaæ je kolejno). W przypadku projektowania za pomoc¹ programu Visual Basic do iterowania s³u¿y polecenie **For Each**. Jeœli u¿ywany jest program Visual C++, wyliczanie elementów kolekcji umo¿liwiaj¹ metody **Item** i **Count**. Dostêp do w³aœciwoœci obiektu daje w³aœciwoœæ **Value**, w której jako parametr nale¿y okreœliæ nazwê w³aœciwoœci.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Tworzenie nowych obiektów programu MTS

W przypadku niektórych kolekcji do tworzenia nowych obiektów mo¿na wykorzystaæ metodê **Add**. Inne kolekcje wymagaj¹ instalowania nowych obiektów (na przyk³ad sk³adników) za pomoc¹ interfejsu narzêdziowego. Szczegó³owe informacje na ten temat mo¿na znaleŸæ pod has³em Korzystanie z narzêdziowych obiektów katalogu w programie MTS.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Zapisywanie zmian obiektów programu MTS

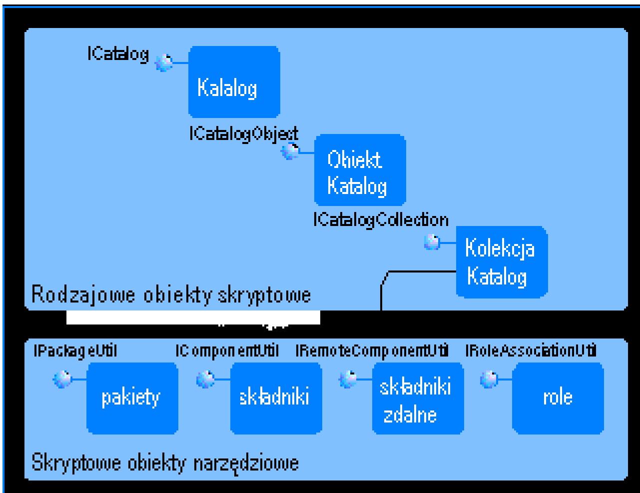
Zmiany w³aœciwoœci, nowe obiekty (tworzone za pomoc¹ metody **Add**) oraz obiekty usuniête (za pomoc¹ metody **Remove**, któr¹ obs³uguje wiêkszoœæ kolekcji) s¹ przechowywane w pamiêci do momentu wywo³ania metody **SaveChanges**. Po wywo³aniu metody **SaveChanges** wszystkie zmiany wprowadzone w obrêbie obiektu CatalogCollection s¹ stosowane do katalogu. Jeœli obiekt CatalogCollection zostanie zwolniony przed wywo³aniem metody **SaveChanges** lub jeœli metody **Populate** lub **PopulateByKe**y zostan¹ wywo³ane przed wywo³aniem metody **SaveChanges**, wszystkie zmiany oczekuj¹ce na zapisanie zostan¹ utracone.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Korzystanie z obiektów narzêdziowych katalogu programu MTS

Warstwa obiektów narzêdziowych zawiera obiekty PackageUtil, ComponentUtil, RemoteComponentUtil i RoleAssociationUtil. Obiekty te s³u¿¹ do wykonywania pewnych operacji bezpoœrednio na katalogu (takich, jak instalacja sk³adników czy instalacja pakietów wstêpnie wbudowanych). Obiekty narzêdziowe s¹ zwi¹zane z konkretnym typem kolekcji. Dostêp do obiektu narzêdziowego zapewnia wywo³anie metody **GetUtilInterface** obiektu CatalogCollection.



Podczas korzystania z danego obiektu narzêdziowego zmiany wprowadzone w katalogu nie maj¹ wp³ywu na obiekt CatalogCollection, z którego pozyskano obiekt narzêdziowy. Aby obejrzeæ zmiany, nale¿y wywo³aæ metodê **Populate** lub **PopulateByKey** obiektu CatalogCollection. Po zastosowaniu metody narzêdziowej nie ma potrzeby wywo³ywania metody **SaveChanges**, poniewa¿ wprowadzone zmiany s¹ zapisywane w katalogu natychmiast.

Zobacz te¿

Obiekty administracyjne programu MTS, Korzystanie z typów kolekcji programu MTS, Metody obiektów administracyjnych programu MTS

Obs³uga b³êdów w katalogu programu MTS

Kolekcja katalogu i metody narzêdziowe katalogu zwracaj¹ parametr HRESULTS, który wskazuje na b³¹d lub brak b³êdu. W programie Visual Basic do przechwytywania b³êdów i uzyskiwania ich kodów s³u¿y obiekt Err i instrukcja obs³ugi b³êdów **On Error**. Metody obs³uguj¹ce wiele obiektów (takie, jak **SaveChanges** i **InstallPackage**) mog¹ przechwytywaæ wiele b³êdów jednoczeœnie (zwi¹zanych z ró¿nymi obiektami). Do zestawu kodów tych b³êdów mo¿na uzyskaæ dostêp za pomoc¹ kolekcji **ErrorInfo**. Kolekcjê tê wywo³uje siê przy u¿yciu metody **GetCollection**. Z ka¿d¹ instancj¹ obiektu CatalogCollection jest skojarzona kolekcja **ErrorInfo**, w której s¹ przechowywane kody b³êdów dla ostatniej metody, której wywo³anie nie powiod³o siê. Podczas instalacji pakietu kolekcji **ErrorInfo** mo¿na u¿yæ do obejrzenia sk³adników, które ju¿ zainstalowano.

Zaleca siê takie oprogramowanie aplikacji, aby po ka¿dym wywo³aniu metody by³a sprawdzana jej poprawnoœæ (b³¹d lub brak b³êdu). W szczególnoœci, podczas dostarczania nazw kolekcji i w³aœciwoœci program powinien sprawdzaæ kod zwrotny parametru E_INVALIDARG (w programie Visual Basic jest to b³¹d czasu wykonywania nr 5). Kod ten wskazuje, ¿e niektóre nazwy kolekcji lub w³aœciwoœci nie s¹ obs³ugiwane.

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Kolekcja ErrorInfo programu MTS

Przyk³ady skryptów napisanych w jêzyku Visual Basic, automatyzuj¹cych administrowanie w programie MTS

Do wywo³ywania skryptowych obiektów administracyjnych s³u¿y dowolny jêzyk programowania zgodny z Automatyzacj¹ OLE (na przyk³ad VBScript). Program MTS zawiera szereg przyk³adowych skryptów administracyjnych, ilustruj¹cych sposób automatyzacji procedur programu MTS Explorer za pomoc¹ obiektów skryptowych.

Aby skrypty Automatyzacji uruchamiaæ na zewn¹trz stron HTML, na komputerze nale¿y zainstalowaæ program Windows Scripting Host (WSH). Program WSH jest narzêdziem do obs³ugi skryptów, uruchamianym z wiersza poleceñ. Mo¿na go zainstalowaæ na komputerze wyposa¿onym w system Windows NT, za pomoc¹ pakietu Windows NT 4.0 Option Pack. Program WSH pozwala wykonywaæ skrypty bezpoœrednio z konsoli poleceñ lub z pulpitu systemu Windows. Skryptów obs³ugiwanych przez program nie trzeba osadzaæ w dokumentach HTML; mo¿na je uruchamiaæ albo bezpoœrednio z konsoli poleceñ, albo klikaj¹c ich pliki na pulpicie. Bardziej szczegó³owe informacje na ten temat zawiera dokumentacja programu Windows Scripting Host.

Aby dowiedzieæ siê, w jaki sposób korzystaæ z obiektów administracyjnych skryptu, nale¿y uruchomiæ jeden ze skryptów przyk³adowych, zapisanych w podkatalogu \program files\mtx\samples\WSH. Podkatalog WSH zawiera nastêpuj¹ce piêæ skryptów (napisanych w jêzyku VB Script):

- InstDLL.vbs
- InstPak.vbs
- Uninst.vbs
- InstDllCLI.vbs
- InstPakCLI.vbs

Powy¿sze skrypty automatyzuj¹ procedury administracyjne dla pakietu Sample Bank (Przyk³adowy bank). Na przyk³ad za pomoc¹ skryptu InstDLL.vbs s¹ wywo³ywane obiekty wykonuj¹ce nastêpuj¹ce czynnoœci: usuniêcie istniej¹cych wersji pakietu Sample Bank, utworzenie nowego pakietu o nazwie Sample Bank, instalacja w nowym pakiecie sk³adników pakietu Sample Bank pochodz¹cych z bibliotek DLL programów Visual Basic, Visual C++ i Visual J++ , zmiana atrybutów transakcji, dodanie nowej roli. Skrypt InstPak.vbs s³u¿y równie¿ do instalacji pakietu Sample Bank, a skrypt Uninst.vbs ma na celu automatyzacjê odinstalowania pakietu Sample Bank w programie MTS. Po uruchomieniu skryptu wyniki jego wykonania mo¿na obejrzeæ w programie MTS Explorer. W tym celu nale¿y klikn¹æ przycisk **Odœwie¿**, znajduj¹cy siê na pasku narzêdzi.

Przed zastosowaniem skryptów InstDLL.vbs, InstPak.vbs i Uninst.vbs nale¿y zmodyfikowaæ okreœlon¹ w skrypcie œcie¿kê pliku tak, aby odnosi³a siê ona do lokalizacji wybranych plików. Na przyk³ad œcie¿ka pliku w skrypcie InstPak.vbs okreœla domyœln¹ lokalizacjê pakietu Sample Bank:

path="C:\Program Files\Samples\Packages\"

Jeœli pakiet Sample Bank jest zainstalowany w innym miejscu, nale¿y zmieniæ powy¿szy wiersz skryptu tak, aby okreœla³ aktualn¹ lokalizacjê pakietu.

Skrypty InstDllCLI.vbs i InstPakCLI.vbs obs³uguj¹ parametry wiersza poleceñ, dziêki czemu uruchamiaj¹c skrypt w oknie konsoli mo¿na okreœliæ lokalizacjê plików za pomoc¹ parametru. W tym celu nale¿y otworzyæ katalog zawieraj¹cy skrypt, a nastêpnie podaæ nazwê skryptu i lokalizacjê pakietu lub bibliotek DLL. Na przyk³ad aby zainstalowaæ pakiet Sample Bank zapisany w katalogu domyœlnym, nale¿y otworzyæ katalog zawieraj¹cy skrypty przyk³adowe oraz wpisaæ w wierszu poleceñ nastêpuj¹ce polecenie:

InstPakCLI.vbs "C:\Program Files\MTX\Samples\Packages\"

Aby móc korzystaæ ze skryptu InstDLL.vbs, nale¿y wczeœniej zarejestrowaæ sk³adniki programu Java z pliku vjacct.dll. W tym celu nale¿y wykonaæ jedn¹ z poni¿szych czynnoœci:

- Zainstalowaæ pakiet, uruchamiaj¹c skrypt InstDLL.vbs
- Zarejestrowaæ sk³adniki programu Java za pomoc¹ Kreatora sk³adników ActiveX programu Microsoft Visual Studio 97.

Zobacz te¿

Instalacja przyk³adowych skryptów administracyjnych programu MTS

Przyk³adowa aplikacja utworzona za pomoc¹ programu Visual Basic, automatyzuj¹ca administrowanie w programie MTS

Przyk³adowa aplikacja, utworzona w programie Visual Basic w wersji 5.0, ma na celu ilustracjê sposobu korzystania z metod obiektów Catalog, CatalogObject i CatalogCollections, automatyzuj¹cych podstawowe czynnoœci administracyjne dla pakietu o nazwie “Scriptable Admin Demo”.

Uwaga Projekt utworzony w jêzyku Visual Basic nale¿y skonfigurowaæ tak, aby odwo³ywa³ siê do biblioteki typów administracyjnych programu MTS (biblioteki typów MTSAdmin). W tym celu na pasku narzêdzi projektu nale¿y zaznaczyæ opcjê **References**. Nastêpnie nale¿y przeszukaæ dostêpne pliki odwo³añ i odszukaæ plik o nazwie “MTS 2.0 Admin Type Library”. W przypadku zmiennych "wi¹zanych póŸniej" (wi¹zanie ustalane po uruchomieniu programu) i uprzedniej rejestracji na komputerze lokalnym pliku MTXADMIN.DLL program Visual Basic zlokalizuje bibliotekê typów bez dalszej konfiguracji.

Aby usun¹æ jakiekolwiek istniej¹ce pakiety o nazwach “Scriptable Admin Demo”

1. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu katalogu.

```
Dim catalog As Object
Set catalog = CreateObject("MTSAdmin.Catalog.1")
```

2. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê **Packages**. Kolekcja **Packages** jest zwracana bez pobierania danych z katalogu, w zwi¹zku z czym pozostanie ona pusta a¿ do momentu zakoñczenia metody **GetCollection**.

```
Dim packages As Object
Set packages = catalog.GetCollection("Packages")
```

3. Odszukaj poprzedni¹ wersjê pakietu "Scriptable Admin Demo", zmuszaj¹c kolekcjê **Packages** do odczytu ze wszystkich pakietów i wyszukiwania pakietów o nazwie "Scriptable Admin Demo". Przeszukuj kolekcjê za pomoc¹ pêtli rozpoczynaj¹cej sie od najwy¿szego indeksu; w pêtli wywo³uj metodê **Remove**. Metoda **Remove** powoduje zwolnienie obiektu, usuniêcie obiektu z kolekcji i przesuniêcie obiektów kolekcji tak, ¿e obiekt o indeksie (*n+1*) staje siê obiektem (*n*) (dla wszystkich *n* wiêkszych lub równych ni¿ indeks obiektu usuwanego). Wynik metody **Remove** jest natychmiastowy. Metody **Item** i **Count,** wywo³ywane po ka¿dym wykonaniu metody **Remove**, powoduj¹ zmiany indeksów. Tym niemniej, usuniêcie pakietu zostanie uwzglêdnione w katalogu dopiero po wykonaniu metody **SaveChanges** (zobacz krok 4).

```
packages.Populate
Dim pack As Object
n = packages.Count
For i = n - 1 To 0 Step -1
    If packages.Item(i).Value("Name") = "Scriptable Admin Demo" Then
        packages.Remove (i)
End If
Next
```

4. Wywo³aj metodê **SaveChanges**, aby zapisaæ zmiany w katalogu danych.

```
packages.SaveChanges
```

Aby utworzyæ nowy pakiet o nazwie “Scriptable Admin Demo Package”

1. Za pomoc¹ metody **Add** dodaj nowy pakiet i zanotuj skojarzony z nim identyfikator. Metoda **Add** spowoduje dodanie obiektu do kolekcji, ale wprowadzone przez ni¹ zmiany zostan¹ zastosowane do katalogu dopiero po wywo³aniu metody

SaveChanges (zobacz krok 3). W wyniku metody **Add** dla wszystkich w³aœciwoœci zostan¹ okreœlone wartoœci domyœlne. Identyfikatorem domyœlnym stanie siê nowy, unikatowy identyfikator.

```
Dim newPack As Object
Dim newPackID As Variant
Set newPack = packages.Add
newPackID = newPack.Value("ID")
```

2 Zaktualizuj w³aœciwoœci **Name** i **SecurityEnabled**.

```
newPack.Value("Name") = "Scriptable Admin Demo"
newPack.Value("SecurityEnabled") = "N"
```

3 Wywo³aj metodê **SaveChanges**, aby zapisaæ nowy pakiet w katalogu. Wartoœci¹ zwrotn¹ wywo³ania jest liczba obiektów, które zmieniono, dodano lub usuniêto. Jeœli nie dokonano ¿adnych zmian, zwracane jest 0.

```
n = packages.SaveChanges
```

Aby zaktualizowaæ w³aœciwoœci pakietu "Scriptable Admin Demo" i pobraæ kolekcjê ComponentsInPackage.

1 Wywo³aj metodê **PopulateByKey**, aby ponownie odczytaæ pakiet z katalogu. Przeka¿ tablicê zawieraj¹c¹ klucze do odczytywanych obiektów. W poni¿szym przyk³adzie kodu u¿yto tablicy zawieraj¹cej jeden element (tj. identyfikator utworzonego przed chwil¹ pakietu).

```
Dim keys(0) as Variant
keys(0) = newPackId
packages.PopulateByKey keys
```

2 Pobierz pakiet (obiekt pakietu) z kolekcji.

```
Dim package As Object
Set package = packages.Item(0)
```

3 Zaktualizuj w³aœciwoœæ **SecurityEnabled** pakietu.

```
package.Value("SecurityEnabled") = "Y"
```

4 Wywo³aj metodê **GetCollection**, aby pobraæ kolekcjê ComponentsInPackage. Jako parametr wywo³ania przeka¿ klucz do pakietu "Scriptable Admin Demo".

```
Set components = packages.GetCollection("ComponentsInPackage",_
package.Key)
```

5 Wywo³aj metodê **SaveChanges**, aby zapisaæ zmiany w katalogu.

```
packages.SaveChanges
```

Aby zainstalowaæ sk³adnik componentasdefcomponent w pakiecie "Scriptable Admin Demo package":

1 Wywo³aj metodê **GetUtilInterface**, aby pobraæ obiekt narzêdziowy sk³adnika. Obiekt ten s³u¿y do instalacji sk³adników.

```
Dim util As Object
Set util = components.GetUtilInterface
On Error GoTo installFailed
```

2 Wywo³aj metodê **InstallComponent**, przekazuj¹c w wywo³aniu ci¹g znaków okreœlaj¹cy nazwê biblioteki DLL (biblioteki do³¹czanej dynamicznie) instalowanego sk³adnika. Jeœli sk³adnik nie zawiera zewnêtrznej biblioteki typów ani *biblioteki DLL typu proxy-stub*, w miejsce drugiego i trzeciego argumentu wywo³ania przeka¿ ci¹gi puste. Po zainstalowaniu nowego sk³adnika nie musisz wywo³ywaæ metody **SaveChanges**. Wszystkie sk³adniki zawarte w bibliotece DLL zostan¹ zainstalowane i natychmiast zapisane do katalogu. Na koniec wywo³aj metodê GetCLSIDs, aby pobraæ identyfikatory klas (CLSID) zainstalowanych sk³adników.

```
Form2.Show 1
Dim thePath As String
thePath = Form2.MTSPath + "\samples\packages\vbacct.dll"
util.InstallComponent thePath, "", ""
Dim installedCLSIDs() as Variant
util.GetCLSIDs thePath, "", installedCLSIDs
On Error GoTo 0
```

3 Wywo³aj metodê **PopulateByKey**, aby z powrotem odczytaæ zainstalowane przed chwil¹ sk³adniki. Warto zauwa¿yæ, ¿e sk³adniki zainstalowane za pomoc¹ metody **InstallComponent** zostan¹ pokazane w kolekcji dopiero po wywo³aniu metody **Populate** lub **PopulateByKey**, s³u¿¹cej do odczytu danych z katalogu.

```
components.PopulateByKey installedCLSIDs
```

Aby odszukaæ sk³adnik Bank.CreateTable i usun¹æ go z pakietu "Scriptable Admin Demo package":

1 Wywo³aj sk³adniki o kolejnych indeksach i zmieñ ich atrybuty transakcji za pomoc¹ metod **Item** i **Count**.

```
Dim component As Object
n = components.Count
For i = n - 1 To 0 Step -1
    Set component = components.Item(i)
    component.Value("Transaction") = "Required"
```

2 Odszukaj i usuñ sk³adnik Bank.CreateTable. Kolekcja jest przegl¹dana wstecz, indeks po indeksie, a metoda **Remove** jest wywo³ywana w pêtli.

```
If component.Value("ProgID") = "Bank.CreateTable" Then
        components.Remove (i)
    End If
Next
```

3 Pobierz now¹ wartoœæ licznika (n) i ponownie wywo³uj kolejne elementy kolekcji. Sk³adnik Bank.CreateTable zostanie usuniêty ze zbioru danych dopiero po wywo³aniu metody **SaveChanges**. Wyœwietl okno komunikatu, informuj¹ce u¿ytkownika o pomyœlnym zakoñczeniu instalacji.

```
n = components.Count
For i = 0 To n - 1
    Set component = components.Item(i)
    Debug.Print component.Value("ProgID")
    Debug.Print component.Value("DLL")
Next

n = components.SaveChanges
MsgBox "Scriptable Admin Demo package installed and configured."
Exit Sub

installFailed:
    MsgBox "Error code " + Str$(Err.Number) + " installing " + thePath + "   Make sure
the MTS path you entered is correct and that vbacct.dll is not already installed."
End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Do instalacji, usuwania i aktualizacji w³aœciwoœci pakietów i sk³adników mo¿na wykorzystaæ skryptowe obiekty administracyjne. W ramach poni¿szych tematów Pomocy przedstawiono czynnoœci programistyczne oraz przyk³adowy kod programu utworzonego w jêzyku Visual Basic w wersji 5.0, ilustruj¹ce zastosowanie metod skryptowych obiektów administracyjnych do automatyzacji nastêpuj¹cych zadañ:

- Automatyzacja instalacji wstêpnie wbudowanego pakietu programu MTS
- Automatyzacja procesu tworzenia nowego pakietu programu MTS oraz instalacji sk³adników
- Automatyzacja wyliczania zainstalowanych pakietów programu MTS w celu uaktualnienia w³aœciwoœci
- Automatyzacja wyliczania zainstalowanych pakietów programu MTS w celu usuniêcia pakietu
- Automatyzacja wyliczania zainstalowanych sk³adników programu MTS w celu usuniêcia sk³adnika

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja instalacji wstêpnie wbudowanego pakietu MTS

Aby zainstalowaæ w programie MTS Explorer wstêpnie wbudowany pakiet "Test.pak":

1 Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas instalacji pakietu wbudowanego.

```
Private Sub InstallPackage_Click()
Dim catalog As Object
Dim packages As Object
Dim util As Object
```

2 Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3 Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê Packages najwy¿szego poziomu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
```

4 Utwórz instancjê obiektu PackageUtil, po czym wywo³aj metodê **InstallPackage**, aby zainstalowaæ pakiet o nazwie "test.pak".

```
Set util = packages.GetUtilInterface
util.InstallPackage "c:\test.pak", "", 0
Exit Sub
```

5 W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)
End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja procesu tworzenia nowego pakietu programu MTS oraz instalacji sk³adników

Aby utworzyæ nowy pakiet o nazwie “My Package” i zainstalowaæ w nim sk³adniki:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas tworzenia nowego pakietu i instalowania sk³adników.

```
Dim catalog As Object
Dim packages As Object
Dim newPack As Object
Dim componentsInNewPack As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz z obiektu CatalogCollection kolekcjê Packages najwy¿szego poziomu. Nastêpnie wywo³aj metodê **Add**, aby dodaæ nowy pakiet.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set newPack = packages.Add
Dim newPackID As String
```

4. Okreœl nazwê pakietu jako “My Package” i zapisz zmiany wprowadzone w kolekcji **Packages**.

```
newPackID = newPack.Key
newPack.Value("Name") = "My Package"
packages.SaveChanges
```

5. Wywo³aj metodê **GetCollection**, aby uzyskaæ dostêp do kolekcji ComponentsInPackage. Nastêpnie utwórz instancjê obiektu ComponentUtil i wywo³aj metodê **InstallComponent**, aby wype³niæ nowy pakiet sk³adnikami.

```
Set componentsInNewPack =
packages.GetCollection("ComponentsInPackage", newPackID)
   Set util = componentsInNewPack.GetUtilInterface
util.InstallComponent"d:\dllfilepath", "", ""
Exit Sub
```

6. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja wyliczania zainstalowanych pakietów programu MTS w celu zaktualizowania w³aœciwoœci

Aby wyliczaæ zainstalowane pakiety w celu zaktualizowania w³aœciwoœci pakietu o nazwie “My Package”:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas wyliczania zainstalowanych pakietów w celu zaktualizowania w³aœciwoœci pakietu.

```
Private Sub BrowseUpdate_Click()
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim componentsInNewPack As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê Packages najwy¿szego poziomu. Nastêpnie wywo³aj metodê **Populate**, aby wype³niæ kolekcjê pakietami z katalogu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Wyliczaj kolejne pakiety kolekcji, aby odnaleŸæ pakiet o nazwie “My Package”. Po odszukaniu pakietu “My Package” zmieñ ustawienie w³aœciwoœci **SecurityEnabled** na “Y”. Wywo³aj metodê **SaveChanges**, aby zapisaæ efekt zaktualizowania w³aœciwoœci pakietu.

```
For Each pack In packages
        If pack.Name = "My Package" Then
            pack.Value("SecurityEnabled") = "Y"
            Exit For
        End If
    Next
packages.SaveChanges

    Exit Sub
```

5. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

 End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja wyliczania zainstalowanych pakietów MTS w celu usuniêcia pakietu

Aby wyliczaæ zainstalowane pakiety w celu usuniêcia pakietu o nazwie “My Package”:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas wyliczania zainstalowanych pakietów w celu usuniêcia konkretnego pakietu.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim componentsInPack As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê **Packages.** Nastêpnie wywo³aj metodê **Populate**, aby wype³niæ kolekcjê pakietami zainstalowanymi w katalogu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Za pomoc¹ metod **Count** i **Item** wyliczaj kolejne pakiety kolekcji, aby odnaleŸæ pakiet o nazwie “My Package”. Po odszukaniu pakietu wywo³aj metodê **Remove**, aby usun¹æ pakiet. Nastêpnie wywo³aj metodê **SaveChanges**, aby zapisaæ efekt usuniêcia.

```
For i = 0 To packages.Count-1
        Set pack = packages.Item(i)
        If pack.Name = "My Package" Then
            packages.Remove (i)
            packages.savechanges
            Exit For
        End If
    Next

    Exit Sub
```

5. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja wyliczania zainstalowanych sk³adników programu MTS w celu usuniêcia sk³adnika

Aby wyliczaæ zainstalowane sk³adniki w celu usuniêcia konkretnego sk³adnika:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas wyliczania zainstalowanych sk³adników w celu usuniêcia konkretnego sk³adnika.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim componentsInPack As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê Packages. Nastêpnie wywo³aj metodê **Populate**, aby wype³niæ kolekcjê pakietami zainstalowanymi w katalogu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Wyliczaj kolejne pakiety kolekcji, aby odnaleŸæ pakiet o nazwie "My Package". Wywo³aj metodê **GetCollection**, aby pobraæ kolekcjê ComponentsInPackage. Za pomoc¹ metody **Populate** wype³nij kolekcjê ComponentInPackages danymi, a nastêpnie wyliczaj kolejne elementy kolekcji w celu odnalezienia sk³adnika "Bank.Account". Po odszukaniu sk³adnika usuñ go wywo³uj¹c metodê **Remove**, po czym zapisz zmiany kolekcji wywo³uj¹c metodê **SaveChanges**.

```
For Each pack In packages
        If pack.Name = "My Package" Then
            Set componentsInPack = packages.GetCollection("ComponentsInPackage", 
pack.Key)
            componentsInPack.Populate
            For i = 0 To componentsInPack.Count
                Set comp = componentsInPack.Item(i)
                If comp.Name = "Bank.Account" Then
                    componentsInPack.Remove (i)
                    componentsInPack.savechanges
                    Exit For
                End If
Next
            Exit For
        End If
    Next

    Exit Sub
```

5. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)
```

```
End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja zaawansowanych czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Skryptowe obiekty administracyjne mo¿na wykorzystaæ do konfigurowania klientów i ról, eksportowania pakietów oraz uzyskiwania dostêpu do nazw kolekcji i w³aœciwoœci obs³ugiwanych przez katalog. W ramach poni¿szych tematów Pomocy przedstawiono czynnoœci programistyczne oraz przyk³adowy kod programu utworzonego w jêzyku Visual Basic w wersji 5.0, ilustruj¹ce zastosowanie metod skryptowych obiektów administracyjnych do automatyzacji nastêpuj¹cych zadañ:

- Automatyzacja dostêpu do nazw kolekcji zwi¹zanych z programem MTS
- Automatyzacja dostêpu do informacji o w³aœciwoœciach programu MTS
- Automatyzacja konfigurowania ról programu MTS
- Automatyzacja eksportowania pakietu programu MTS
- Automatyzacja konfigurowania klienta programu MTS w celu korzystania ze sk³adników zdalnych
- Automatyzacja aktualizacji w³aœciwoœci pakietu programu MTS na serwerach zdalnych

Automatyzacja dostêpu do nazw kolekcji zwi¹zanych z programem MTS

Kolekcja RelatedCollectionInfo zawiera listê zwi¹zanych ze sob¹ kolekcji, do których mo¿na uzyskaæ dostêp z danej kolekcji. Listê w³aœciwoœci obs³ugiwanych przez kolekcjê RelatedCollectionInfo mo¿na znaleŸæ w podrêczniku MTS Administrative Reference pod has³em RelatedCollectionInfo.

Aby uzyskaæ dostêp do kolekcji zwi¹zanych z dan¹ kolekcj¹ i wyœwietliæ ich nazwy:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas uzyskiwania dostêpu do obiektu dostarczaj¹cego nazwy kolekcji zwi¹zanych z dan¹ kolekcj¹

```
Dim catalog As Object
Dim packages As Object
Dim RelatedCollectionInfo As Object
Dim collName As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection** pobierz kolekcjê Packages. Nastêpnie wywo³aj metodê **GetCollection** obiektu kolekcji Packages, aby uzyskaæ dostêp do kolekcji. Warto zauwa¿yæ, ¿e podczas wywo³ywania metody **GetCollection** (zapewniaj¹cej dostêp do kolekcji RelatedCollectionInfo) wartoœæ klucza pozostawiono pust¹. Wartoœæ ta nie jest wykorzystana, poniewa¿ informacje zawarte w kolekcji RelatedCollectionInfo maj¹ byæ to¿same dla wszystkich pakietów. Na koniec wywo³aj metodê **Populate**, aby wype³niæ kolekcjê RelatedCollectionInfo informacjami z katalogu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set RelatedCollectionInfo =_ packages.GetCollection("RelatedCollectionInfo", "")
RelatedCollectionInfo.Populate
```

4. Wyliczaj kolejne elementy kolekcji i wyœwietlaj nazwy poszczególnych kolekcji.

```
For Each collName In RelatedCollectionInfo
    Debug.Print collName.Name
Next

Exit Sub
```

5. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt Err.

```
failed:
      MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja dostêpu do informacji o w³aœciwoœciach programu MTS

Kolekcja PropertyInfo zawiera informacje o poszczególnych w³aœciwoœciach kolekcji. Wiêcej informacji o tej kolekcji mo¿na znaleŸæ w podrêczniku MTS Administrative Reference pod has³em PropertyInfo.

Aby uzyskaæ dostêp do informacji o w³aœciwoœciach i wyœwietliæ listê poszczególnych w³aœciwoœci kolekcji:

1 Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas uzyskiwania dostêpu do informacji o w³aœciwoœciach zapisanych w katalogu.

```
Dim catalog As Object
Dim packages As Object
Dim propertyInfo As Object
Dim property As Object
```

2 Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3 Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection** pobierz kolekcjê **Packages**. Nastêpnie wywo³aj metodê **GetCollection** kolekcji **Packages**, aby uzyskaæ dostêp do kolekcji PropertyInfo. Warto zauwa¿yæ, ¿e podczas wywo³ywania metody **GetCollection** (zapewniaj¹cej dostêp do kolekcji PropertyInfo) wartoœæ klucza pozostawiono pust¹. Wartoœæ ta nie jest wykorzystana, poniewa¿ informacje zawarte w kolekcji PropertyInfo maj¹ byæ to¿same dla wszystkich pakietów. Na koniec wywo³aj metodê **Populate**, aby wype³niæ kolekcjê PropertyInfo informacjami z obiektu Catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set propertyInfo = packages.GetCollection("PropertyInfo", "")
propertyInfo.Populate
```

4 Wyliczaj kolejne elementy kolekcji i wyœwietl listê nazw kolejnych w³aœciwoœci w kolekcji.

```
For Each property In propertyInfo
        Debug.Print property.Name
    Next

    Exit Sub
```

5 W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt Err.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja konfigurowania ról programu MTS

Aby skonfigurowaæ role dla pakietu i sk³adnika oraz przydzieliæ u¿ytkownikowi uprawnienia administracyjne:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas konfigurowania roli dla konkretnego sk³adnika.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim comp As Object
Dim newUser As Object
Dim newRole As Object
Dim componentsInPack As Object
Dim RolesInPackage As Object
Dim usersInRole As Object
Dim rolesForComponent As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê Packages. Nastêpnie wype³nij kolekcjê Packages danymi z <u>katalogu</u>.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Wyliczaj elementy kolekcji Packages w celu odnalezienia pakietu o nazwie “My Package”. Po odszukaniu pakietu “My Package” wywo³aj metodê **GetCollection**, aby uzyskaæ dostêp do kolekcji RolesInPackage. Za pomoc¹ metody **Add** dodaj do pakietu now¹ rolê. Nadaj roli nazwê “R1” i zapisz zmiany kolekcji.

```
    If pack.Name = "My Package" Then
  Set rolesInPack = packages.GetCollection("RolesInPackage", pack.Key)
            Set newRole = rolesInPack.Add
            newRole.Value("Name") = "R1"
            rolesInPack.savechanges
```

5. Wywo³aj metodê **GetCollection** kolekcji RolesInPackage, aby pobraæ kolekcjê UsersInRole. Za pomoc¹ funkcji **Add** dodaj do roli istniej¹cego u¿ytkownika systemu NT. Nadaj u¿ytkownikowi nazwê “administrator”. Zapisz zmiany kolekcji UsersInRole.

```
            Set usersInRole = RolesInPackage.GetCollection("UsersInRole",
        newRole.Key)
            Set newUser = usersInRole.Add
            newUser.Value("User") = "administrator"
            usersInRole.savechanges
```

6. Za pomoc¹ metody **GetCollection** pobierz kolekcjê ComponentsInPackage. Wype³nij kolekcjê ComponentsInPackage danymi, a nastêpnie wyliczaj kolejne elementy kolekcji w celu odnalezienia sk³adnika Bank.Account. Aby skojarzyæ ze sk³adnikiem now¹ rolê, za pomoc¹ metody **GetUtilInterface** utwórz instancjê obiektu RoleAssociationUtil. Nastêpnie wywo³uj¹c metodê **AssociateRole**, skojarz now¹ rolê ze sk³adnikiem.

```
Set componentsInPack = packages.GetCollection("ComponentsInPackage", pack.Key)
```

```
            componentsInPack.Populate
            For Each comp In componentsInPack
                If comp.Name = "Bank.Account" Then
                    Set rolesForComponent =
componentsInPack.GetCollection("RolesForPackageComponent", comp.Key)
                    Set util = rolesForComponent.GetUtilInterface
                    util.associateRole (newRole.Key)
                    Exit For
                End If
            Next
            Exit For
        End If
    Next

    Exit Sub
```

7 W razie nieudanej instalacji pakietu, wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt Err.

```
    failed:
    MsgBox "Failure code " + Str$(Err.Number)

    End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja eksportowania pakietu programu MTS

Aby wyeksportowaæ pakiet o nazwie "test.pak":

1 Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas eksportu pakietu.

```
Dim catalog As Object
Dim packages As Object
Dim util As Object
```

2 Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3 Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection**, pobierz kolekcjê Packages. Wywo³aj metodê **Populate**, aby wype³niæ pakiet danymi z katalogu.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4 Wyliczaj elementy kolekcji Packages w celu odnalezienia pakietu o nazwie "My Package". Po odszukaniu pakietu utwórz instancjê obiektu narzêdziowego pakietu i wywo³aj metodê **ExportPackage**.

```
For Each pack In packages
        If pack.Name = "My Package" Then
            Set util = packages.GetUtilInterface
            util.ExportPackage pack.Key, "c:\test.pak", 0
            Exit For
        End If
    Next

Exit Sub
```

5 W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt Err.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja konfigurowania klienta programu MTS w celu korzystania ze sk³adników zdalnych

Aby skonfigurowaæ klienta do korzystania ze sk³adnika zdalnego Bank.CreateTable:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas konfigurowania klienta (korzystaj¹cego z programu MTS) dla potrzeb uruchamiania sk³adników zdalnych.

```
Dim catalog As Object
Dim remoteComps As Object
Dim util As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³uj¹c metodê **GetCollection** pobierz kolekcjê RemoteComponents. Nastêpnie za pomoc¹ metody **GetUtilInterface** utwórz instancjê obiektu RemoteComponentUtil. Aby zainstalowaæ sk³adnik zdalny, wywo³aj metodê **InstallRemoteComponentByName**, a nastêpnie podaj nazwê komputera serwera, nazwê pakietu na serwerze i nazwê sk³adnika.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set remoteComps = catalog.GetCollection("RemoteComponents")
Set util = remoteComps.GetUtilInterface
    util.InstallRemoteComponentByName "remote1", "New","Bank.CreateTable"
Exit Sub
```

4. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt **Err**.

```
    failed:
    MsgBox "Failure code " + Str$(Err.Number)

    End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

Automatyzacja aktualizacji w³aœciwoœci pakietu programu MTS na serwerach zdalnych

Aby uaktualniæ w³aœciwoœci pakietu w pamiêci komputera zdalnego o nazwie “remote1”:

1. Zadeklaruj obiekty, które bêd¹ wykorzystywane podczas konfigurowania klienta (korzystaj¹cego z programu MTS) w celu rozmieszczania sk³adników zdalnych i administrowania nimi.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim root As Object
```

2. Za pomoc¹ polecenia **On Error** obs³u¿ b³êdy czasu wykonywania, otrzymywane w razie zwrotu przez metodê awaryjnej wartoœci HRESULT. Polecenie **On Error** i obiekt **Err** pozwalaj¹ sprawdziæ przechwytywalne b³êdy oraz odpowiednio na nie zareagowaæ.

```
On Error GoTo failed
```

3. Wywo³aj metodê **CreateObject**, aby utworzyæ instancjê obiektu Catalog. Wywo³aj metodê **Connect**, aby uzyskaæ dostêp do ***kolekcji g³ównej*** komputera o nazwie “remote1”. Kolekcja g³ówna jest kolekcj¹ zapewniaj¹c¹ dostêp do wszystkich kolekcji najwy¿szego poziomu danego komputera. Kolekcja ta nie zawiera obiektów ani nie ma w³aœciwoœci. Warto zauwa¿yæ, ¿e podczas wywo³ywania z kolekcji g³ównej metody **GetCollection** nie u¿yto wartoœci klucza. Wywo³aj metodê **GetCollection**, aby pobraæ z komputera zdalnego kolekcjê Packages. Nastêpnie za pomoc¹ metody **Populate** wype³nij kolekcjê danymi.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set root = catalog.Connect("remote1")
Set packages = root.GetCollection("Packages", "")
packages.Populate
```

4. Ustaw w³aœciwoœæ SecurityEnabled pakietu “My Package” jako “Y” i zapisz zmiany wprowadzone w kolekcji.

```
 For Each pack In packages
        If pack.Name = "My Package" Then
            pack.Value("SecurityEnabled") = "Y"
            Exit For
        End If
    Next
    packages.savechanges

    Exit Sub
```

5. W razie nieudanej instalacji pakietu wyœwietlaj komunikat o b³êdzie. W komunikacie wykorzystaj obiekt Err.

```
    failed:
    MsgBox "Failure code " + Str$(Err.Number)

    End Sub
```

Zobacz te¿

Obiekty administracyjne programu MTS, Typy kolekcji programu MTS, Metody obiektów administracyjnych programu MTS, Automatyzacja czynnoœci administracyjnych programu MTS za pomoc¹ programu Visual Basic

MTS Overview and Concepts

How Does MTS Work?

Explains the elements of Microsoft Transaction Server and how they work together to provide the infrastructure and programming model to develop and deploy components.

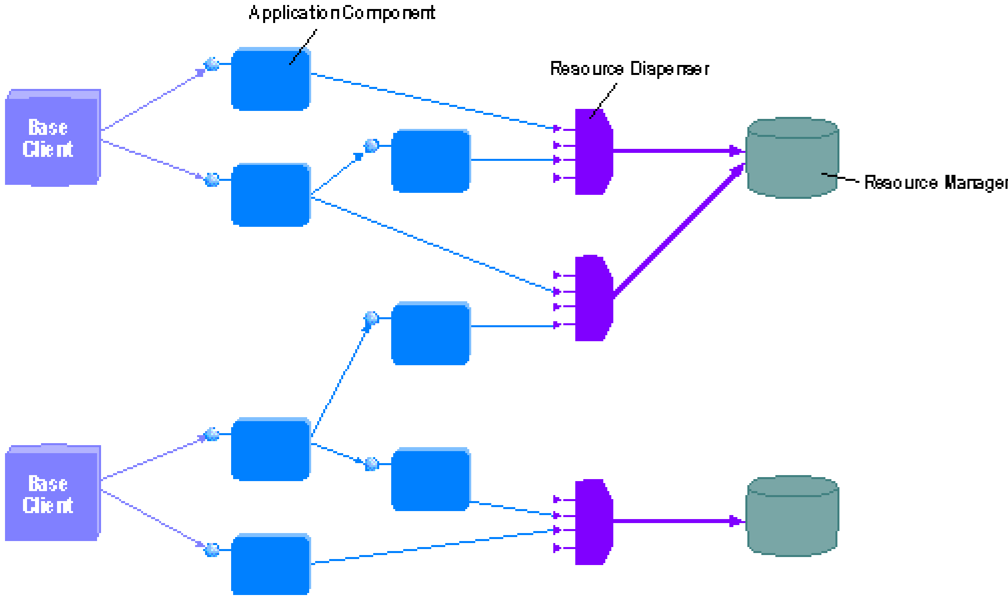
MTS Programming Concepts

Explains the key concepts that a component application developer needs to understand for developing applications.

How Does MTS Work?

This topic describes the elements of Microsoft Transaction Server (MTS) and explains how they provide the infrastructure and programming model to develop and deploy MTS components.

Elements of MTS



Contents

Application Components

MTS Executive

Server Processes

Resource Managers

Resource Dispensers

Microsoft Distributed Transaction Coordinator

MTS Explorer

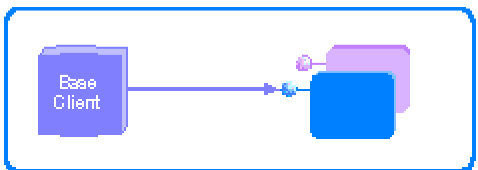
Application Components

Application components model the activity of a business. These components implement the business rules, providing views and transformations of the application state. Consider, for example, the case of an online bank. Records in one or more database systems represent the durable state of the business, such as the amount of money in an account. The application components update that state to reflect such changes as debits and credits.

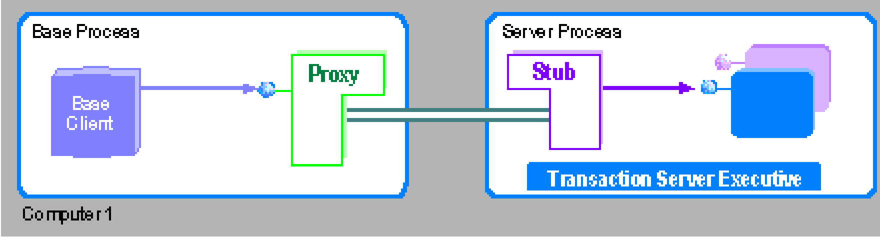
MTS shelters developers from complex server issues, allowing them to focus on implementing business functions. Because components running in the MTS run-time environment can take advantage of transactions, developers can write applications as if they run in isolation. MTS handles the concurrency, resource pooling, security, context management, and other system-level complexities. The transaction system, working in cooperation with database servers and other types of resource managers, ensures that concurrent transactions are atomic, consistent, have proper isolation, and that, once committed, the changes are durable. For more information on the benefits of transactions, see Transactions.

MTS also makes it easier to build distributed applications by providing location transparency. MTS automatically loads the component into a process environment. An MTS component can be loaded into a client application process (in-process component), or into a separate surrogate server process environment, either on the client's computer (local component) or on another computer (remote component).

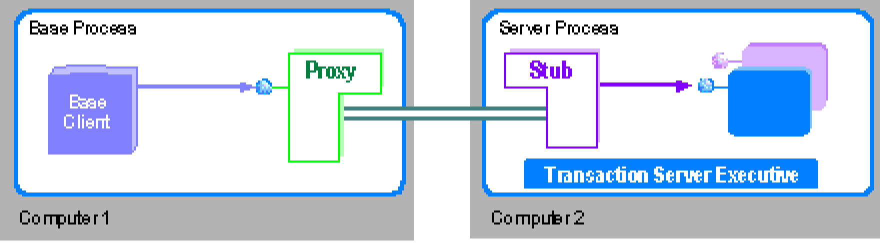
In-process, local, and remote components



In-Process Application Component



Local Application Component



Remote Component

MTS Components and COM

MTS components are COM in-process server components contained in (DLLs). They are distinguished from other COM components in that they execute in the MTS run-time environment. You can create and implement these components with Visual Basic, Visual C++, Visual J++, or any ActiveX-compatible development tool.

Note that the term *component* represents the code that implements a COM object. For example, Visual C++ components are implemented as classes. Likewise, Visual Basic components are implemented by class modules.

MTS imposes specific requirements on components beyond those required by COM (see MTS Component Requirements). This allows MTS to provide services to the component that would not otherwise be possible. These include increased scalability and robustness and simplified system management.

See Also

Base Clients vs. MTS Components, MTS Objects, Business Logic in MTS Components, Server Processes

MTS Executive

The MTS Executive is a dynamic-link library (DLL) that provides run-time services for MTS components, including thread and context management. This DLL loads into the processes that host application components and runs in the background.

MTS also provides a set of resource dispensers that simplify access to shared resources in a server process. For more information, see Resource Dispensers.

See Also

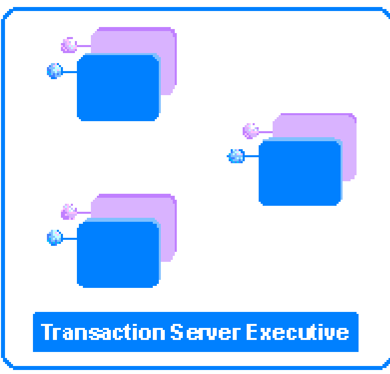
Application Components

Server Processes

A *server process* is a system process that hosts the execution of an application component. A server process can host multiple components and can service tens, hundreds, or potentially thousands of clients. You can configure multiple server processes to execute on a single computer. Each server process reflects a separate trust boundary and fault isolation domain.

Other process environments can also host application components. As a result, you can deploy applications that meet varying distribution, performance, and fault isolation requirements. For example, you can configure MTS components to be loaded directly into Microsoft Internet Information Server (IIS). You can also configure them to load directly into client processes.

Server Process



Resource Managers

A *resource manager* is a system service that manages durable data. Server applications use resource managers to maintain the durable state of the application, such as the record of inventory on hand, pending orders, and accounts receivable. Resource managers work in cooperation with the Microsoft Distributed Transaction Coordinator to guarantee atomicity and isolationto an application.

MTS supports resource managers, such as Microsoft SQL Server version 6.5, that implement the OLE Transactions protocol.

See Also

Resource Dispensers, Microsoft Distributed Transaction Coordinator

Resource Dispensers

A *resource dispenser* manages nondurable shared state on behalf of the application components within a process. Resource dispensers are similar to resource managers, but without the guarantee of durability. MTS provides two resource dispensers:

- The ODBC resource dispenser
- The Shared Property Manager

ODBC Resource Dispenser

The ODBC resource dispenser manages pools of database connections for MTS components that use the standard ODBC interfaces, allocating connections to objects quickly and efficiently. Connections are automatically enlisted on an object's transactions, and the resource dispenser can automatically reclaim and reuse connections. The ODBC 3.0 Driver Manager is the ODBC resource dispenser; the Driver Manager DLL is installed with MTS.

Shared Property Manager

The Shared Property Manager provides synchronized access to application-defined, process-wide properties (variables). For example, you can use it to maintain a Web-page hit counter or to maintain the shared state for a multiuser game.

See Also

ISharedPropertyGroupManager Interface, Creating a Simple ActiveX Component, Sharing State

Microsoft Distributed Transaction Coordinator

The Microsoft Distributed Transaction Coordinator (MS DTC) is a system service that coordinates transactions. Work can be committed as an atomic transaction even if it spans multiple resource managers, potentially on separate computers.

MS DTC was first released as part of Microsoft SQL Server version 6.5 and is included in MTS, providing a low-level infrastructure for transactions. MS DTC implements a two-phase commit protocol to ensure that the transaction outcome (either commit or abort) is consistent across all resource managers involved in a transaction. MS DTC ensures atomicity, regardless of failures.

See Also

Resource Managers

MTS Explorer

You can use the MTS Explorer to deploy application components. You can also use it to view and manipulate items in the MTS run-time environment.

For a complete discussion of using the MTS Explorer for application administration, see the *Administrator's Guide*.

MTS Programming Concepts

This topic explains the key concepts that a component application developer needs to understand to develop Microsoft Transaction Server (MTS) applications.

Contents

Transactions

MTS Objects

MTS Clients

Activities

Components and Threading

Programmatic Security

Error Handling

Transactions

MTS simplifies the task of developing application components by allowing you to perform work with transactions. This protects applications from anomalies caused by concurrent updates or system failures.

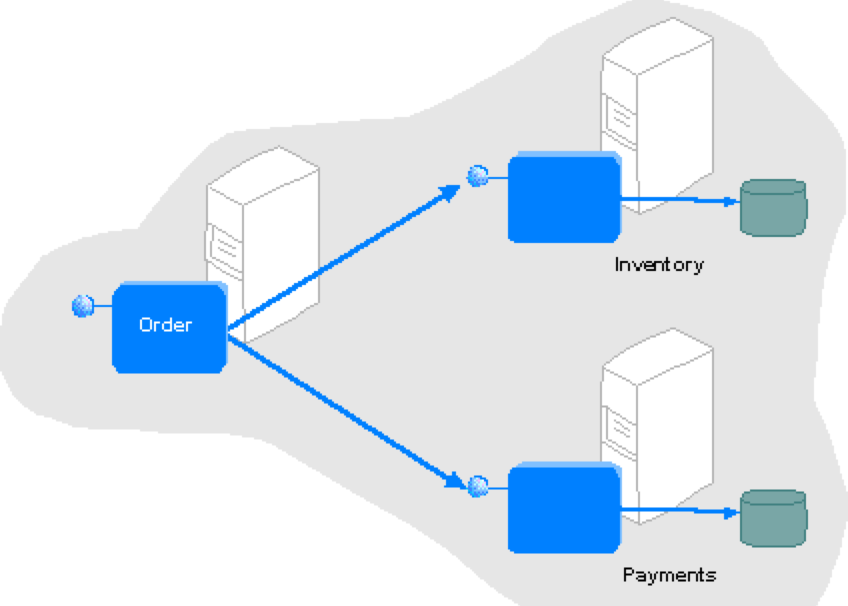
Transactions maintain the ACID properties:

- *Atomicity* ensures that all the updates completed under a specific transaction are committed and made durable, or that they get aborted and rolled back to their previous state.
- *Consistency* means that a transaction is a correct transformation of the system state, preserving the state invariants.
- *Isolation* protects concurrent transactions from seeing each other's partial and uncommitted results, which might create inconsistencies in the application state. Resource managers use transaction-based synchronization protocols to isolate the uncommitted work of active transactions.
- *Durability* means that committed updates to managed resources, such as a database record, survive failures, including communication failures, process failures, and server system failures. Transactional logging even allows you to recover the durable state after disk media failures.

The intermediate states of a transaction are not visible outside the transaction, and either all the work happens or none of it does. This allows you to develop application components as if each transaction executes sequentially and without regard to concurrency. This is a tremendous simplification for application developers.

You can declare that a component is transactional, in which case MTS associates transactions with the component's objects. When an object's method is executed, the services that resource managers and resource dispensers perform on its behalf execute under a transaction. This can also include work that it performs for other MTS objects. Work from multiple objects can be composed into a single atomic transaction.

Multiple objects composed into a transaction



Without transactions, error recovery is extremely difficult, especially when multiple objects update multiple databases. The possible combinations of failure modes are too great even to consider. Transactions simplify error recovery. Resource managers automatically undo the transaction's work, and the application retries the entire business transaction.

Transactions also provide a simple concurrency model. Because a transaction's isolation prevents one client's work from interfering with other clients, you can develop components as though only a single client executes at a time.

Components Declare Transactional Requirements

Every MTS component has a transaction attribute that is recorded in the MTS catalog. MTS uses this attribute during object creation to determine whether the object should be created to execute within a transaction, and whether a transaction is required or optional. For more information on transaction attributes, see Transaction Attributes.

Components that make updates to multiple transactional resources, such as database records, for example, can ensure that their objects are always created within a transaction. If the object is created from a context that has a transaction, the new context inherits that transaction; otherwise, the system automatically initiates a transaction.

Components that only perform a single transactional update can be declared to support, but not require, transactions. If the object is created from a context that has a transaction, the new context inherits that transaction. This allows the work of multiple objects to be composed into a single atomic transaction. If the object is created from a context that does not have a transaction, the object can rely on the resource manager to ensure that the single update is atomic.

How Work Is Associated with a Transaction

An object's associated context object indicates whether the object is executing within a transaction and, if so, the identity of the transaction.

Resource dispensers can use the context object to provide transaction-based services to the MTS object. For example, when an object executing within a transaction allocates a database connection by using the ODBC resource dispenser, the connection is automatically enlisted on the transaction. All database updates using this connection become part of the transaction, and are either atomically committed or aborted. For more information, see Enlisting Resources in Transactions.

Stateful and Stateless Objects

Like any COM object, MTS objects can maintain internal state across multiple interactions with a client. Such an object is said to be stateful. MTS objects can also be stateless, which means the object does not hold any intermediate state while waiting for the next call from a client.

When a transaction is committed or aborted, all of the objects that are involved in the transaction are deactivated, causing them to lose any state they acquired during the course of the transaction. This helps ensure transaction isolation and database consistency; it also frees server resources for use in other transactions.

Completing a transaction enables MTS to deactivate an object and reclaim its resources, thus increasing the scalability of the application. Maintaining state on an object requires the object to remain activated, holding potentially valuable resources such as database connections. Stateless objects are more efficient and are thus recommended. For more information on object deactivation, see Deactivating Objects.

How Objects Can Participate in Transaction Outcome

You can use methods implemented on the **IObjectContext** interface to enable an MTS object to participate in determining a transaction's outcome. The **SetComplete**, **SetAbort**, **DisableCommit**, and **EnableCommit** methods work in conjunction with the component's transaction attribute to allow one or more objects to be composed simply and safely within transactions.

- **SetComplete** indicates that the object has successfully completed its work for the transaction. The object is deactivated upon return from the method that first entered the context.

- **SetAbort** indicates that the object's work can never be committed. The object is deactivated upon return from the method that first entered the context.
- **EnableCommit** indicates that the object's work is not necessarily done, but that its transactional updates can be committed in their current form.
- **DisableCommit** indicates that the object's transactional updates can not be committed in their current form.

Both **SetComplete** and **SetAbort** deactivate the object on return from the method. MTS reactivates the object on the next call that requires object execution.

Objects that need to retain state across multiple calls from a client can protect themselves from having their work committed prematurely by the client. By calling **DisableCommit** before returning control to the client, the object can guarantee that its transaction cannot successfully be committed without the object doing its remaining work and calling **EnableCommit**.

Client-Controlled vs. Automatic Transactions

Transactions can either be controlled directly by the client, or automatically by the MTS run-time environment.

Clients can have direct control over transactions by using a transaction context object. The client uses the **ITransactionContext** interface to create MTS objects that execute within the client's transactions, and to commit or abort the transactions.

Transactions can automatically be initiated by the MTS run-time environment to satisfy the component's transaction expectations. MTS components can be declared so that their objects always execute within a transaction, regardless of how the objects are created. This feature simplifies component development, because you do not need to write application logic to handle the special case where an object is created by a client not using transactions.

This feature also reduces the burden on client applications. Clients do not need to initiate a transaction simply because the component that they are using requires it.

MTS automatically initiates transactions as needed to satisfy a component's requirements. This event occurs, for example, when a client that is not using transactions creates an object in an MTS component that is declared to require transactions.

MTS completes automatic transactions when the MTS object that triggered their creation has completed its work. This event occurs when returning from a method call on the object after it has called **SetComplete** or **SetAbort**. **SetComplete** causes the transaction to be committed; **SetAbort** causes it to be aborted.

A transaction cannot be committed while any method is executing in an object that is participating in the transaction. The system behaves as if the object disables the commit for the duration of each method call.

See Also

Building Transactional Components, Multiple Transactions

Transaction Attributes

Every MTS component has a transaction attribute. The transaction attribute can have one of the following values:

- **Requires a transaction.** This value indicates that the component's objects must execute within the scope of a transaction. When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction, MTS automatically creates a new transaction for the object.
- **Requires a new transaction.** This value indicates that the component's objects must execute within their own transactions. When a new object is created, MTS *automatically* creates a new transaction for the object, regardless of whether its client has a transaction.
- **Supports transactions.** This value indicates that the component's objects can execute within the scope of their client's transactions. When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction, the new context is also created without one.
- **Does not support transactions.** This value indicates that the component's objects do not run within the scope of transactions. When a new object is created, its object context is created without a transaction, regardless of whether the client has a transaction.

Most MTS components are declared as either **Supports transactions** or **Requires a transaction**. These values allow an object to execute within the scope of its client's transaction. You can see the difference between these values when an object is created from a context that does not have a transaction. If the component's transaction attribute is **Supports transactions**, the new object runs without a transaction. If it is declared as **Requires a transaction**, MTS automatically initiates a transaction for the new object.

Declaring a component as **Requires a new transaction** is similar to using **Requires a transaction** in that the component's objects are guaranteed to execute within transactions. However, when you declare the transaction attribute this way, an object never runs inside the scope of its client's transaction. Instead, the system always creates independent transactions for the new objects. For example, you can use this for auditing components that record work done on behalf of another transaction regardless of whether the original transaction commits or aborts.

Specifying **Does not support transactions** ensures that an object's context does not contain a transaction. This value is the default setting and is primarily used with versions of COM components that precede MTS.

Setting the Transaction Attribute

The transaction attribute is part of a component definition. The component developer determines it, and changes to it are not recommended.

You can set a transaction attribute at development time by:

- Using values defined in Mtxattr.h. You can specify these values in an .odl file to encode them into the component's type library.
- Using the MTS Explorer to create a package file for deploying your components.

Because Because Microsoft Visual Basic automatically generates a type library, Visual Basic developers must use the MTS Explorer to set the transaction attribute.

See Also

Application Components, Transactions

Enlisting Resources in Transactions

MTS provides automatic transactions, which means that transaction requirements are declared as component properties. MTS automatically begins transactions and commits or aborts these transactions on behalf of the component, based on the component's transaction property.

Automatic transactions work because resource dispensers pass transactions to the resource manager. For example, the ODBC Driver Manager is a resource dispenser for ODBC database connections. When a database connection is requested from a transactional component, the ODBC Driver Manager obtains the transaction from the object's context. The ODBC Driver Manager then associates (enlists) the database connection with the transaction.

If you do not have a resource dispenser, you can build your own using the Microsoft Transaction Server Beta Software Development Kit (SDK), available at http://www.microsoft.com/support/transactions/. For more information, see the *Resource Dispenser Guide* and the samples included with the MTS Beta SDK.

For a resource manager to participate in an MTS transaction, it must support one of the following protocols:

- OLE Transactions
- The X/Open DTP XA standard

OLE Transactions, the object-oriented, *two-phase commit* protocol defined by Microsoft, is the preferred protocol. OLE Transactions is based on the Component Object Model (COM) and is used by resource managers in order to participate in distributed transactions coordinated by Microsoft Distributed Transaction Coordinator (DTC). OLE Transactions is supported by Microsoft SQL Server version 6.5. For more information on supporting OLE Transactions, see the *Resource Manager Guide* and the samples included with the MTS Beta SDK.

XA is the two-phase commit protocol defined by the X/Open DTP group. XA is natively supported by many Unix databases, including Informix, Oracle, and DB2.

For MTS to work with XA-compliant resource managers, an OLE Transactions-to-XA mapper, provided by the MTS Beta SDK, makes it relatively straightforward for XA-compliant resource managers to provide resource dispensers that can accept OLE Transactions from MTS and translate to XA. For more information, see "Mapping OLE Transactions to the XA Protocol" in the MTS Beta SDK, or contact your resource manager vendor.

See Also

Resource Dispensers, Transactions

Determining Transaction Outcome

This section discusses how applications determine whether a transaction will commit or abort.

First, it is important to understand that the MTS objects involved in a transaction do not need to know the transaction outcome. All objects involved in a transaction are automatically deactivated. Deactivation causes the objects to lose any state that they acquired during the transaction. Consequently, their behavior is not affected by the outcome of the transaction.

Each object can participate in determining the outcome of a transaction. Objects call **SetComplete**, **SetAbort**, **EnableCommit**, and **DisableCommit,** based on the desired component behavior. For example, an object would typically call **SetAbort** after receiving an error from a database operation, on a method call to another object, or due to a violation of a business rule such as an overdrawn account.

The client of the transaction determines its success or failure (commit or abort) based on values returned from the method call that caused the transaction to complete. The client can be either a base client or another MTS object that exists outside the transaction. The client must know which methods cause transactions to complete and how the method output values can be used to determine success (commit) or failure (abort).

An object method that intends to commit a transaction typically returns an HRESULT value of S_OK after calling **SetComplete**. On return, MTS automatically completes the transaction. If the transaction commits, the S_OK value is returned to the client. If it aborts, the HRESULT value is changed to CONTEXT_E_ABORTED. The client can use these two values to determine the outcome.

An object method typically notifies its client that it has forced the transaction to abort by calling **SetAbort** in one of two ways:

- Return S_OK and use an output parameter to indicate the failure.
- Return an HRESULT error code. Different codes could be used to distinguish different causes, or the generic CONTEXT_E_ABORTED error could be used.

For example, the Sample Bank application uses an output parameter to indicate failure:

```
Public Function Perform(lngPrimeAccount As Long, _
   lngSecondAccount As Long, lngAmount As Long, _
   strTranType As String, ByRef strResult As String) _
   As Long

' get our object context
Dim ctxObject As ObjectContext
Set ctxObject = GetObjectContext()

On Error GoTo ErrorHandler

' check for security
If (lngAmount > 500 Or lngAmount < -500) Then
    If Not ctxObject.IsCallerInRole("Managers") Then
        Err.Raise Number:=ERROR_NUMBER, _
        Description:="Need 'Managers' role " + _
           "for amounts over $500"
    End If
End If
.
.
.
ctxObject.SetComplete        ' we are finished and happy
Perform = 0
```

```
Exit Function

ErrorHandler:

ctxObject.SetAbort           ' we are unhappy
strResult = Err.Description  ' return the error message
Perform = -1           ' indicate that an error occured

End Function
```

It is also important to note that there are failure scenarios where the client cannot determine the transaction outcome. This situation results, for example, when a call failure occurs due to a transport error such as RPC_E_CONNECTION_TERMINATED). In such cases, it is necessary to use an application-defined protocol to determine the transaction outcome.

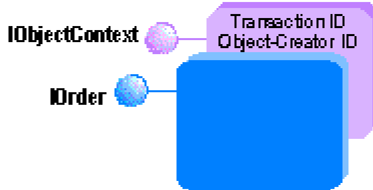
On clustered servers, MTS will not automatically reconnect to MS DTC in the event of a failover. Not enough information exists about the transaction composition and state to determine the appropriate course of action. Retries remain the responsibility of the client application. The client cannot differentiate an error caused by failover from other errors.

Resource managers are guaranteed to get transaction outcomes as part of the two-phase commit protocol managed by the Microsoft Distributed Transaction Coordinator. This feature allows resource managers to manage locks and to determine whether it is necessary to make state changes permanent or to discard them.

MTS Objects

An MTS object is an instance of an MTS component. MTS maintains context for each object. This context, which is implicitly associated with the object, contains information about the object's execution environment, such as the identity of the object's creator and, optionally, the transaction encompassing the work of the object. The object context is similar in concept to the process context that an operating system maintains for an executing program.

An MTS object and its associated context object



An MTS object and its associated context object have corresponding lifetimes. MTS creates the context before it creates the MTS object. MTS destroys the context after it destroys the MTS object.

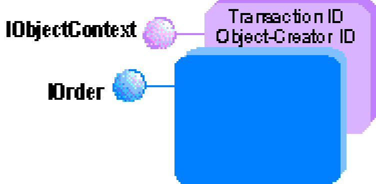
See Also

Application Components, MTS Component Requirements, Context Objects, asconCreatingTransactionServerObjects, Passing Object References, Deactivating Objects

Context Objects

Each MTS object has an associated context object. A *context object* is an extensible MTS object that provides context for the execution of an instance, including transaction, activity, and security properties. When an MTS object is created, MTS automatically creates a context object for it. When the MTS object is released, MTS automatically releases the context object.

An MTS object and its associated context object



An MTS object's context has intrinsic properties that are determined during object creation. These properties include the identity of the client that initiated the object's creation and whether or not the object executes within the scope of a transaction.

The properties established for the new object context are determined by a combination of:

- The transaction attributes of the component as specified in the MTS catalog.
- The properties of the context from which the new object is created. For example, the client's context may contain a transaction.

If your application uses Microsoft Internet Information Server (IIS), you can retrieve IIS intrinsic objects as follows:

- Using Visual Basic, by calling the **Item** method of the context object.
- Using Microsoft Visual C++ or Microsoft Visual J++, by calling the **GetProperty** method of the **IGetContextProperties** interface.

For more information on IIS intrinsic objects, see the IIS documentation.

Contexts Are Implicit

MTS maintains an implicit relationship between an MTS object and its context. This feature eliminates the need for you to pass explicitly a context object through your application.

You can access an MTS object's context by calling the **GetObjectContext** function. This function returns a reference to the **IObjectContext** interface. Resource dispensers and other context-aware services can also access the object's context. This permits the ODBC resource dispenser automatically to enlist connections on the object's transaction.

Before a method of an MTS object is dispatched for execution, that object's context becomes the current context for the thread. This context remains current as long as the object remains within the context. Calling a method in a different context causes that context to become current; the caller's context is automatically restored on return from the method.

Managing References to the Context Object

You must not pass a reference to the context object outside the object. You must explicitly release every reference that you acquire on the object context.

The context object is not available during calls to the component's class factory. This means, for example, that a Visual C++ class implementation cannot access a context object during calls to a constructor or destructor. Objects that require access to the context object during initialization or destruction should implement **IObjectControl**. For more information, see Deactivating Objects.

See Also

MTS Objects, **GetObjectContext**, **IObjectContext**

Creating MTS Objects

You can create MTS objects by:

- Using context objects.
- Using transaction context objects.
- Using standard COM functions like **CoCreateInstance** or **CreateObject**.

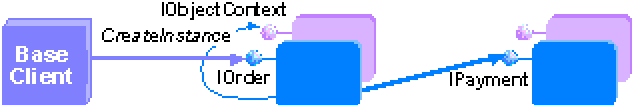
Note If you are using Visual C++ and running an MTS component in-process, you must:

- Call **CoInitialize(**NULL**)** before requesting services from MTS or creating an MTS object.
- Call **CoInitializeSecurity** to initialize process-specific security. MTS security is disabled when loading an MTS object in-process.
- Call **CoUninitialize** only after you have finished using MTS or MTS objects, preferably just prior to terminating your application. You cannot call **CoInitialize** again and invoke more MTS services. Once **CoUninitialize** has been called, your application no longer executes in the MTS run-time environment.

Creating Objects Using a Context Object

You can create an MTS object by calling the **CreateInstance** method on the **IObjectContext** interface of an object's context object. The new MTS object's context inherits the activity, possibly a transaction, and all security identities from the creating object's context.

Creating an object using a context object



Creating Objects Using a Transaction Context Object

If you want your base client to control transaction boundaries, use a transaction context object. You can create an MTS object by calling the **CreateInstance** method of the **ITransactionContext** interface. The new MTS object's context inherits the activity, possibly a transaction, and the identity of the initial client from the transaction context object. You can call the **Commit** method to commit an object's work and the **Abort** method to abort its work.

Creating an object using a transaction context object

Creating Objects Using CoCreateInstance

You can create MTS objects by using **CoCreateInstance** or any equivalent method based on **CoGetClassObject** and **IClassFactory::CreateInstance**. While this approach should suffice for many base client applications, there are some significant limitations for the client, including the inability to control transaction boundaries. Base clients that need this additional level of control can use a transaction context object.

Instantiating an object with CoCreateInstance

When you use **CoGetClassObject** with a component that is registered to run under MTS, it returns a reference to an MTS-provided class factory. This allows MTS to participate in the client's calls to

IClassFactory::**CreateInstance**. The MTS class factory creates the context object and then calls the component's real class factory.

For clustered servers, if you are using the **CoCreateInstanceEx** function, use the name of the virtual server containing the MSDTC resource in the *pwszName* field of the COSERVERINFO structure. (See the Microsoft Platform SDK documentation for more details about **CoCreateInstanceEx**.)

Important It is recommended that you do not call **CoCreateInstance** to create MTS objects from within MTS objects. When you do so, the new object's context cannot inherit any properties from its client's context. In particular, the new object cannot execute within the scope of its client's transaction.

Aggregation

You cannot use an MTS object as part of an aggregate of other objects. **CoCreateInstance** returns CLASS_E_NOAGGREGATION to indicate an attempt to create an MTS object with another controlling **IUnknown**.

You can, however, create an MTS object that is implemented as an aggregation of objects.

Creating Objects Using Visual Basic

You can use the following object creation methods in Microsoft Visual Basic to create MTS objects:

- The **CreateObject** function
- The **GetObject** function
- The **New** keyword (see the **Important** note for limitations)
- Automatically if the object is an Application object

Using Visual Basic object creation methods results in the same limitations as using **CoCreateInstance**. To inherit a transaction from the creating object's context, use **CreateInstance**.

Important Do not use the **New** operator, or a variable declared **As New**, to create an instance of a class that is part of the active project. In this situation, Visual Basic uses an implementation of object creation that does not use COM. To prevent this occurrence, it is recommended that you mark all objects passed out from a Visual Basic componentas **Public Creatable**, or its equivalent, and created with either the **CreateObject** function or the **CreateInstance** method of the **ObjectContext** object.

See Also

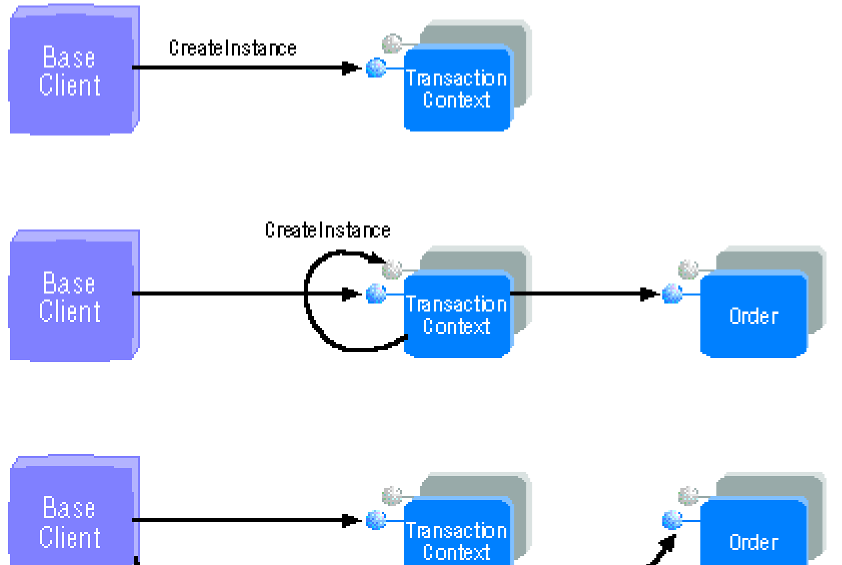
MTS Objects, Calling MTS Components, Context Objects, Passing Object References, Deactivating Objects, **CreateInstance**

Transaction Context Objects

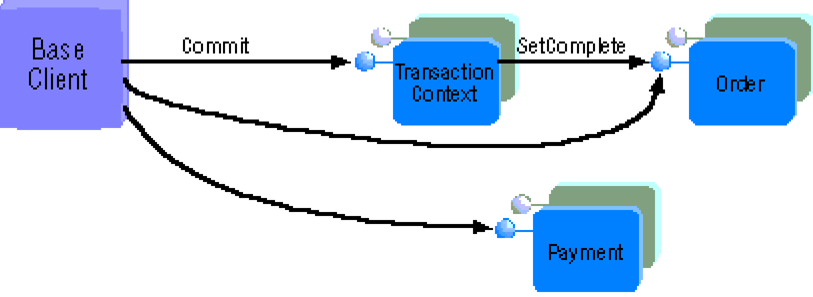
The *transaction context object* allows base clients to combine the work of multiple MTS objects into a single transaction, without having to develop a new component specifically for that purpose.

The transaction context object's methods use its context object as follows:

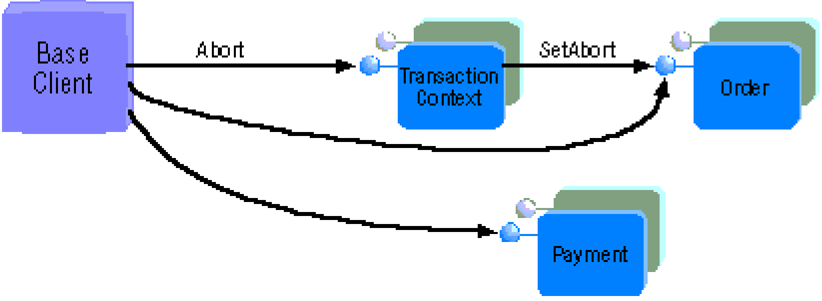
- **CreateInstance** — Calls **CreateInstance** and returns a reference to the newly created object.



- **Commit** — Calls **SetComplete** and returns.



- **Abort** — Calls **SetAbort** and returns.



The transaction context component is defined as **Requires a New Transaction**. You cannot use the

transaction context object to enlist in an existing transaction.

For example, suppose you have two components, Walk and ChewGum. Each component is defined as **Supports Transactions** and calls **SetComplete** when it is finished with its work. A base client could compose the work done by each component in a single transaction.

```
Dim objTxCtx As TransactionContext
Dim objWalk As MyApp.Walk
Dim objChewGum As MyApp.ChewGum

' Get TransactionContext
Set objTxCtx = _
   CreateObject("TxCtx.TransactionContext")

' Create instances of Walk and ChewGum
Set objWalk = _
   objTxCtx.CreateInstance("MyApp.Walk")
Set objChewGum = _
   objTxCtx.CreateInstance("MyApp.ChewGum")

' Both components do work
objWalk.Walk
objChewGum.ChewGum

' Commit the transaction
objTxCtx.Commit
```

Transaction Context Object Limitations

Note the following limitations when using a transaction context object:

- Transaction composition
- Location transparency
- Base client does not have context

Transaction Composition

When using a transaction context object, the application logic that composes the work into a single transaction is tied to a specific base client implementation and the advantages of using MTS components are lost. These implementations include:

- Ability to reuse the application logic as part of an even larger transaction
- Imposition of declarative security
- Flexibility to run the logic remotely from the client

Location Transparency

The transaction context object runs *in-process* with the base client, which means that MTS must exist on the base client computer. This may not be a problem, for example when the transaction context object is used from an Active Server Page (ASP) that is running on the same server as MTS.

Base Client Does Not Have Context

You do not get a context for the base client when you create a transaction context object. Transactional work can only be done indirectly, through MTS objects created by using the transaction context object. In particular, the base client cannot use MTS *resource dispensers* (such as ODBC) and have the work included in of the transaction. For example, developers may be familiar with the following syntax for doing transactional work on relational database systems:

```
BEGIN TRANSACTION
   DoWork
COMMIT TRANSACTION
```

Using the transaction context object in a similar way does not yield the desired result:

```
Set objTxCtx = CreateObject("TxCtx.TransactionContext")
   DoWork
   objTxCtx.Commit
Set objTxCtx = Nothing
```

The call to DoWork in this example will not be enlisted in a transaction. You must build an MTS component that calls DoWork, create an object instance of that component using the transaction context object, then call that object from the base client in order for the work to be part of the client-controlled transaction.

See Also

Transactions, **TransactionContext** Object

Passing Parameters

This topic covers the following:

- Parameter types and marshaling
- Objects as parameters
- Passing large data

Parameter Types and Marshaling

MTS object interfaces must be able to be marshaled. Marshaling interfaces allows calls across thread, process, and machine boundaries.

Standard COM marshaling is used. This means MTS object interfaces must either:

- Have method parameters which are Automation data types and be described in a type library, or
- Use custom interfaces with a MIDL-generated proxy-stub DLL.

For more information on type libraries and proxy-stub DLLs, see MTS Component Requirements.

Custom marshaling is not used. Even if a component supports the **IMarshal** interface, its **IMarshal** methods are never called by the MTS run-time environment.

VBScript Parameters

Components that are intended for use from Active Server Pages (ASPs) using Microsoft Visual Basic® Scripting Edition (VBScript) should support **IDispatch** and limit method parameter types as follows:

- **VBScript version 1.0—**Any Automation type may be passed by value, but not by reference. Method return values must be of type **Variant**.
- **VBScript version 2.0—**Same as VBScript version 1.0, except parameters of type **Variant** may now be passed by reference.

Objects as Parameters

Whether an object is passed *by value* or *by reference* is not specified by the client, but is a characteristic of the object itself. Basic COM objects can either be passed by reference or by value, depending on their implementation. If the COM object uses standard marshaling, then it is passed by reference. COM objects can also implement **IMarshal** to copy data by value. MTS objects are always passed by reference.

Additionally, the function of the object affects how it should be passed as a parameter. When deciding whether to pass objects by value or by reference, it is useful to classify the objects as follows:

- **Recordset Objects—**Encapsulate raw data, such an ADO recordset. Recordset objects are not registered as MTS objects.
- **Business Object—**Encapsulate business logic; for example, an order-processing component. Business objects should be registered as MTS objects.

The following table describes when to pass recordset objects by value or by reference:

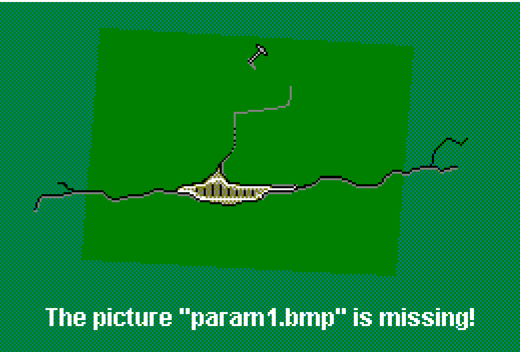
Pass Parameter	If	Client Requirements
By value	Data is relatively small	Recipient requires all data and can get data without reaccessing caller.
By reference	Data is relatively large	Recipient does not require all data and must reaccess caller, possibly many times.

Note Whether data is "small" or "large" also depends on the speed of the connection. For example, if the component is accessed over a corporate intranet, a much larger recordset can be passed to the client in one call than in a call made by a client accessing the component on an Internet server over a modem.

Because business objects are MTS objects, they are always passed by reference.

Passing Large Data

When returning a large amount of data, consider using a Microsoft Active Data Objects (ADO) **Recordset** object. In particular, the Microsoft Advanced Data Connector (ADC) provides a recordset implementation that can be disconnected from the server and marshaled by value to the client.



The disconnected recordset moves state to the client, allowing server resources to be freed. The client can make changes to the recordset and reconnect to the server to submit updates. For more information on state, see Holding State in Objects.

Another method of packaging large amounts of data is to use *safe arrays*. For example, when using Microsoft Remote Data Objects (RDO), you can use the **rdoResultSet.GetRows** method to copy rows into an array, and then pass the array back to the client. This requires fewer calls and is more efficient than issuing **MoveNext** calls across the network for each row.

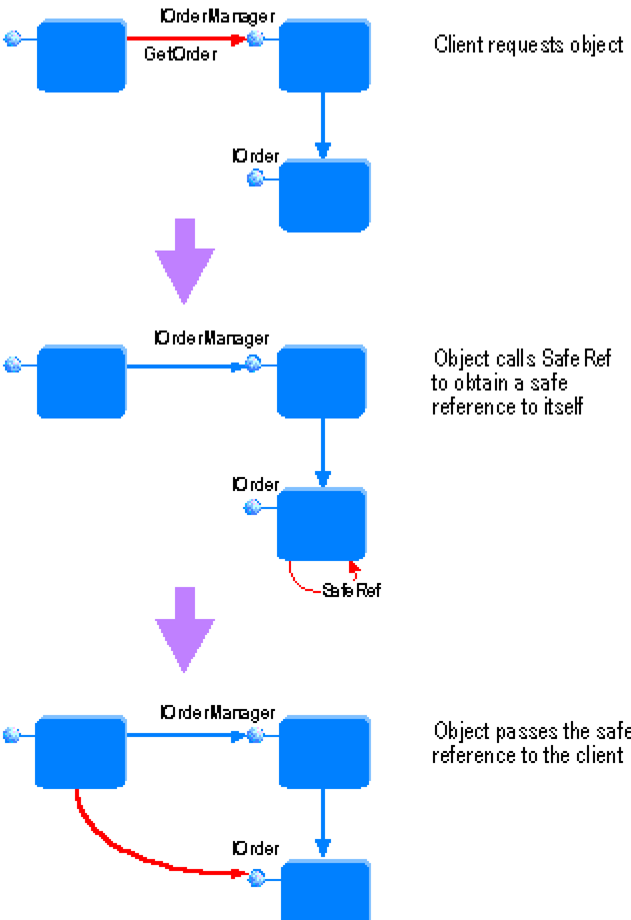
Passing Object References

You must ensure that MTS object references are only exchanged in the following ways:

- Through return from an object creation interface, such as **CoCreateInstance** (or its equivalent), **ITransactionContext::CreateInstance**, or **IObjectContext::CreateInstance**.
- Through a call to **QueryInterface**.
- Through a method that has called **SafeRef** to obtain the object reference.

An object reference that is obtained in these ways is called a *safe reference*. MTS ensures that methods invoked using safe references execute within the correct context.

Using SafeRef to pass a reference to an object



Note It is not safe to exchange references by any other means. In particular, do not pass interfaces outside the object by using global variables. These restrictions are similar to those imposed by COM for references passed between apartments.

Calls that use safe references always pass through the MTS run-time environment. This allows MTS to manage context switches and allows MTS objects to have lifetimes that are independent of client references. For more information, see Deactivating Objects.

Callbacks

It is possible to make *callbacks* to clients and to other MTS components. For example, you can have an object that creates another object. The creating object can pass a reference to itself to the created object; the created object can then use this reference to call the creating object.

If you choose to use callbacks, note the following restrictions:

- Calling back to the base client or another package requires Access-level security on the client. Additionally, the client must be a DCOM server.
- Intervening firewalls may block calls back to the client.
- Work done on the callback executes in the environment of the object being called. It may be part of the same transaction, a different transaction, or no transaction.
- The creating object must call **SafeRef** and pass the returned reference to the created object in order to call back to itself.

See Also

SafeRef

Deactivating Objects

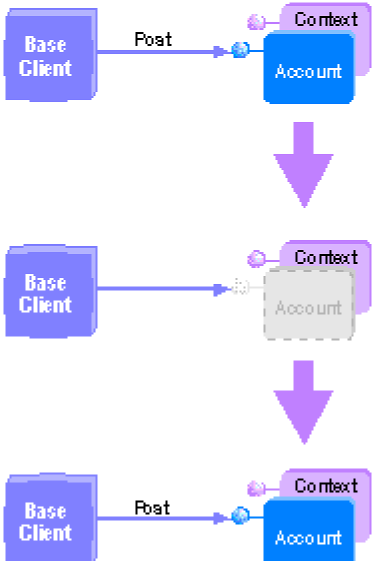
MTS extends COM to allow object deactivation, even while client references are maintained. This makes server applications more scalable by allowing server resources to be used more efficiently.

MTS objects are initially created in the deactivated state. When a client invokes a method on an object that is in a deactivated state, MTS automatically activates the object. During activation, the object is put into its initial state.

Note For Visual C++ developers, calls to **QueryInterface**, **AddRef**, or **Release** do not cause activation.

This ability for an object to be deactivated and reactivated while clients hold references to it is called *just-in-time activation*. From the client's perspective, only a single instance of the object exists from the time the client creates it to the time it is finally released. In actuality, it is possible that the object has been deactivated and reactivated many times.

Just-in-time activation



The context object exists for the entire lifetime of its MTS object, even across one or more deactivation and reactivation cycles.

Object deactivation allows clients to hold references for long periods of time with limited consumption of server resources. Consider, for example, a client application that spends 99 percent of its time between transactions. In this case, the MTS objects are activated less than 1 percent of the time.

When is an object deactivated?

An MTS object is deactivated when any of the following occurs:

- The object requests deactivation.

  An object can request deactivation by using the **IObjectContext** interface. You can use the **SetComplete** method to indicate that the object has successfully completed its work and that the internal object state doesn't need to be retained for the next call from the client. Similarly, **SetAbort** indicates that the object cannot successfully complete its work and that its state does not need to be retained.

  You can develop stateless objects by using MTS objects that deactivate on return from every method.
- A transaction is committed or aborted.

MTS does not allow an object to maintain private state that it acquired during a transaction. When an object's transaction is committed or aborted, the object is deactivated. Of these deactivated objects, the only ones that can continue to exist are the ones that have references from clients outside the transaction. Subsequent calls to these objects reactivate them and cause them to execute in the next transaction.

- The last client releases the object.

  This occurrence is listed here for completeness. The object is deactivated and never reactivated. The object's context is also released.

How are objects deactivated?

MTS deactivates an object by releasing all its references to the object. This causes properly developed components to destroy the object; this feature also requires the component to follow the MTS reference passing rules (see Passing Object References) and the COM reference counting rules.

Note MTS writes an Informational message to the event log when objects that do not report their reference count are deactivated.

Application components are responsible for releasing object resources on deactivation. This includes:

- Resources that are allocated with MTS resource dispensers, such as ODBC database connections.
- All other resources, including references to other objects (including MTS objects and context objects) and memory held by any instances of the component, such as using **delete this** in C++).

Doing Additional Work on Activation and Deactivation

If an object is not already activated and it supports **IObjectControl**, MTS calls the **Activate** method prior to initiating the client request. Components can use the **Activate** method to initialize objects. This is especially important for initialization that requires access to the context object. Here, keep in mind that the context is not available during calls to the component's class factory. Having access to the context object through **Activate** allows you to pass a reference to the context object to other methods; this reference can then be released in the **Deactivate** method. The **Activate** method is also useful for objects that support pooling (see Object Pooling and Recycling).

For objects that support the **IObjectControl** interface, MTS calls the **Deactivate** method when it deactivates the object. You can use this method to free resources held by the object. The **Deactivate** method is also useful for objects that support pooling. Like the **Activate** method, the **Deactivate** method has access to the object context.

See Also

Building Scalable Components, Stateful Components, Object Pooling and Recycling, **SetComplete**, **SetAbort**, **IObjectControl**

Object Pooling and Recycling

Objects that support the **IObjectControl** interface can participate in object recycling and pooling, which can increase the efficiency of activation and deactivation. After MTS calls the **Deactivate** method, it calls the **CanBePooled** method, allowing the object to be pooled for reuse. If the object returns TRUE, the object is added to the object pool. Objects that return FALSE or that do notsupport the **IObjectControl** interface are destroyed.

On activation, MTS uses an object from the pool if one is available. Only if the pool is empty will it use the component's class factory to create a new object.

Components that support object pooling must ensure that an object activated using an object from the pool is indistinguishable to the client from an object that is activated by creating a new object. Component developers must provide the appropriate code in the **Activate** and **Deactivate** method implementations to ensure this behavior.

The following table summarizes MTS run-time actions for client call processing.

	IObjectControl implemented by component	
	No	**Yes**
Step 1		
Activate the object if needed (just-in-time activation).	Use the component class factory to create an object.	Allocate an object from the pool. If pooling is not supported or the pool is empty, then use the component class factory to create an object. Call the object's **Activate** method.
Step 2		
Execute the call.	Call the object method.	Call the object method.
Step 3		
Deactivate the object if requested (**SetComplete** or **SetAbort** called before return).		Call object's **Deactivate** method
	Release the last reference held by the MTS run-time environment.	If the system supports pooling, then call **CanBePooled**. If it returns TRUE, then add the object to the pool. Otherwise, release the last reference held by the MTS run-time environment.

Important Object pooling and recycling is not available in this release. MTS calls **CanBePooled** as described here, but no pooling takes place. This forward-compatibility encourages developers to use **CanBePooled** in their applications now in order to benefit from a future release without having to modify their applications later.

See Also

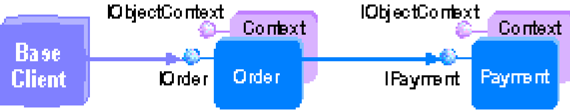
Deactivating Objects, **IObjectControl**

MTS Clients

An application or object that uses an MTS object is referred to as a *client* of the object. It is important to understand that this is a relative term, and describes a *relationship* with a specific object. For example, when MTS object A uses MTS object B, the object A, while still an object, is also a client.

Clients that run outside the direct control of the MTS run-time environment are referred to as *base clients*.

Clients and base clients: The Order object is a client of the Payment object



See Also

Base Clients, Base Clients vs. MTS Components

Base Clients

Base clients are the primary consumers of MTS objects, although base clients execute outside of the MTS run-time environment. One common role of a base client is to provide the application's user interface, and to map the end-user's requests to the business functions exposed by the MTS components.

You can create base clients with a variety of programming languages, including Microsoft Visual C++, Microsoft Visual Basic, Microsoft Visual J++, COBOL, and even Transact SQL. Base client programs can execute in a variety of environments, from application processes to general system services such as Microsoft Internet Information Server (IIS) or SQL Server. In fact, you can use any programming environment in which you can create COM objects and invoke methods on them.

See Also

MTS Clients, Base Clients vs. MTS Components

Base Clients vs. MTS Components

The following table contrasts MTS components with base client applications.

MTS components	Base clients
MTS components are contained in COM dynamic-link libraries (DLLs); MTS loads DLLs into processes on demand.	Base clients can be written as executable files (.exe) or dynamic-link libraries (.dll); MTS is not involved in their initiation or loading.
MTS manages server processes that host MTS components.	MTS does not manage base client processes.
MTS creates and manages the threads used by components.	MTS does not create or manage the threads used by base client applications.
Every MTS object has an associated context object. MTS automatically creates, manages, and releases context objects.	Base clients do not have implicit context objects. They can use transaction context objects, but they must explicitly create, manage, and release them.
MTS objects can use resource dispensers. Resource dispensers have access to the context object, allowing acquired resources to be automatically associated with the context.	Base clients cannot use resource dispensers.

See Also

MTS Clients, Application Components

Activities

An *activity* is a set of objects executing on behalf of a base client application. Every MTS object belongs to one activity. This is an intrinsic property of the object and is recorded in the object's context. The association between an object and an activity cannot be changed. An activity includes the MTS object created by the base client, as well as any MTS objects created by that object and its descendants. These objects can be distributed across one or more processes, executing on one or more computers.

For example, an online banking application may have an MTS object dispatch credit and debit requests to various accounts, each represented by a different object. This dispatch object may use other objects as well, such as a receipt object to record the transaction. This results in several MTS objects that are either directly or indirectly under the control of the base client. These objects all belong to the same activity.

MTS tracks the flow of execution through each activity, preventing inadvertent parallelism from corrupting the application state. This feature results in a single logical thread of execution throughout a potentially distributed collection of objects. By having one logical thread, applications are significantly easier to write.

Whenever you use **CoCreateInstance** or its equivalent to create an MTS object, a new activity is created; note that this includes a base client creating a transaction context object.

When an MTS object is created from an existing context, using either a transaction context object or an MTS object context, the new object becomes a member of the same activity. In other words, the new context inherits the activity identifier of the context used to create it.

MTS only allows a single logical thread of execution within an activity. This is similar in behavior to a COM apartment, except that the objects can be distributed across multiple processes. When a base client calls into an activity, all other requests for work in the activity (such as from another client thread) are blocked until after the initial thread of execution returns back to the client.

Callbacks and Reentrancy

While MTS does not allow multiple threads of execution within an MTS object, reentrancy is possible via callbacks. Suppose you have an object that creates another object. If the creating object passes a reference to itself to the created object, either directly or indirectly, cycles can occur in the call graph. MTS objects that do this must be prepared to receive a method invocation while blocked waiting for a call to complete. MTS ensures that the incoming call belongs to the same logical thread by using the COM logical thread identifier. COM uses the logical thread identifier for a similar purpose in apartment objects.

Limitation

While no parallel execution can exist within the activity on any individual computer, the MTS run-time environment does not protect against clients entering into the same activity through objects on two different computers. This can result in two parallel threads of execution on different computers. However, if the thread of execution on one computer calls an object in the same activity on the other, the call will be blocked.

This behavior is based on the belief that the cost outweighs the benefits of providing fully distributed activity protection, both in terms of development and run-time performance.

See Also

Components and Threading, Passing Object References

Components and Threading

The MTS run-time environment manages threads for you. MTS components need not, and, in fact, should not, create threads. Components must never terminate a thread that calls into a DLL.

Every MTS component has a **ThreadingModel** Registry attribute, which you can specify when you develop the component. This attribute determines how the component's objects are assigned to threads for method execution. You can view the threading-model attribute in the MTS Explorer by clicking the Property view in the Components folder. The possible values are Single, Apartment, and Both.

Single-Threaded Components

All objects of a single-threaded component execute on the main thread. This is compatible with the default COM threading model, which is used for components that do not have a **ThreadingModel** Registry attribute.

The main threading model provides compatibility with COM components that are not reentrant. Because objects always execute on the main thread, method execution is serialized across all objects in the component. In fact, method execution is serialized across all components in a process that uses this policy. This allows components to use libraries that are not reentrant, but it has very limited scalability.

Limitations for Single-Threaded Components

Single-threaded, stateful components are prone to deadlocks. You can eliminate this problem by using stateless objects and calling **SetComplete** before returning from any method.

The following scenario describes how such a deadlock can occur. Suppose you have a single-threaded Account component, which is written to be both transactional and stateful. The following scenario describes how two clients, Client 1 and Client 2, could call objects in a way that causes an application deadlock:

- Client 1 creates Account object A and calls it to update an account record. The database update is done under object A's transaction (Transaction 1). Because the object does not call **SetComplete** before returning to the client, Transaction 1 remains active.
- Next, Client 2 creates Account object B and calls it to update the same account. Because its work is done under a different transaction (Transaction 2), the attempt to update the account record will block while waiting for Transaction 1 to complete.
- Client 1 makes another call to object A, for instance, to have it make another change to the account record and complete the transaction. However, the call must wait for the main thread, which is still busy servicing the call from Client 2.
- The two clients are now deadlocked. Client 1 holds a lock on the account record, while waiting for the server's main thread so that it can complete Transaction 1. Client 2 holds the server's main thread, while waiting for Transaction 1 to complete so that it can update the account record.

Note that this is not a deadlock from the SQL perspective because A's and B's connections are in different transactions.

Apartment-Threaded Components

Each object of an apartment-threaded component is assigned to a thread its *apartment*, for the life of the object; however, multiple threads can be used for multiple objects. This is a standard COM concurrency model. Each apartment is tied to a specific thread and has a Windows message pump.

The apartment threading model provides significant concurrency improvements over the main threading model. Activities determine apartment boundaries; two objects can execute concurrently as long as they are in different activities. These objects may be in the same component or in different

components.

See Also

Activities

Programmatic Security

The MTS security model consists of declarative security and programmatic security. Developers can build both declarative and programmatic security into their components prior to deploying them on a Windows NT security domain.

Important Security is not supported on Windows 95. Note the following application behavior when running MTS on Windows 95:

- All identities are mapped to "Windows 95".
- Role configuration is not supported.
- Checking roles always return success.

Roles are central to the MTS security model. A role is an abstraction that defines a logical group of users. At development time, you use roles to define declarative authorization and programmatic security logic. At deployment time, you bind these roles to specific groups and users.

You can administer package security with the MTS Explorer. This is a form of declarative security, which does no't require any component programming is based on standard Windows NT security.

MTS also allows component applications to implement additional access control programmatically. MTS security is integrated with DCOM and Windows NT security. See the Microsoft Platform SDK for further information on COM security APIs.

See Also

Basic Security Methods, Advanced Security Methods

Basic Security Methods

The **IObjectContext** interface provides two methods for basic programmatic security:

- **IsCallerInRole**
- **IsSecurityEnabled**

The key to understanding how MTS security works is to understand roles, as discussed in the following sections.

Roles from a Development Perspective

A role is a symbolic name that defines a logical group of users for a package of components. For example, an online banking application might define roles for Manager and Teller.

You can define authorization for each component and component interface by assigning roles. For example, in the online banking application, only the Manager may be authorized to perform bank transactions above a certain amount of money.

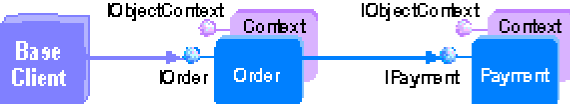
Roles are defined during application development. These roles are then assigned to specific users at deployment time.

Important Roles on a dual interface are not enforced when **IDispatch** (late-binding) is used.

Checking If a Caller Is in a Role

The **IsCallerInRole** method determines if a caller is assigned to a role. The caller is the direct caller, which is the identity of the process (base client or server process) calling into the current server process.

The following illustration shows an application used to order supplies for a business.



You can use roles to determine whether the base client has access to objects in the server process. In this scenario, the server process would check to see if the base client is allowed to place an order. Calling **IsCallerInRole** on the Order object context checks if the direct caller, which is in this case the base client, is in a given role. Such a role might be Purchaser, to restrict the placing of orders to employees within that role.

Security checks are made when a process boundary is crossed. If the Payment object accesses a database, the access rights to the database are derived from the identity of the server process, not the base client. The database would use its own proprietary authorization checking.

Server-process security does not use impersonation. **IsCallerInRole** has the same semantics regardless of how many calls have taken place within the server process. The identity of the direct caller is always used to make the check. For more information on impersonation, see Advanced Security Methods.

Security for In-Process Components

Because the level of trust is process-wide, running MTS components in-process is not recommended for secure applications. Access checks are not made on calls between components in the same server process. Configuring MTS components to run in-process with the base client gives the base client access to all components within that server process.

The **IsSecurityEnabled** method determines if security checking is enabled. This method returns FALSE when running in-process. **IsSecurityEnabled** can be a useful check to make before using

IsCallerInRole. **IsCallerInRole** will always return TRUE when called on an object that is running in-process, which may have unintended effects.

When an MTS component is part of a Library package (in-process), it effectively becomes part of the hosting Server package that creates it. If you create Library packages with components that call **IsCallerInRole**, you should instruct installers of your Library packages to define the Library package's roles on the hosting Server package. Otherwise, **IsCallerInRole** will always fail.

See Also

IsCallerInRole, **IsSecurityEnabled**, Secured Components

Advanced Security Methods

MTS objects can use the **ISecurityProperty** interface to obtain security-related information from the object context, including the identity of the client that created the object, as well as the identity of the current calling client. Applications can use this information to implement custom access control, such as using the Win32 security interfaces.

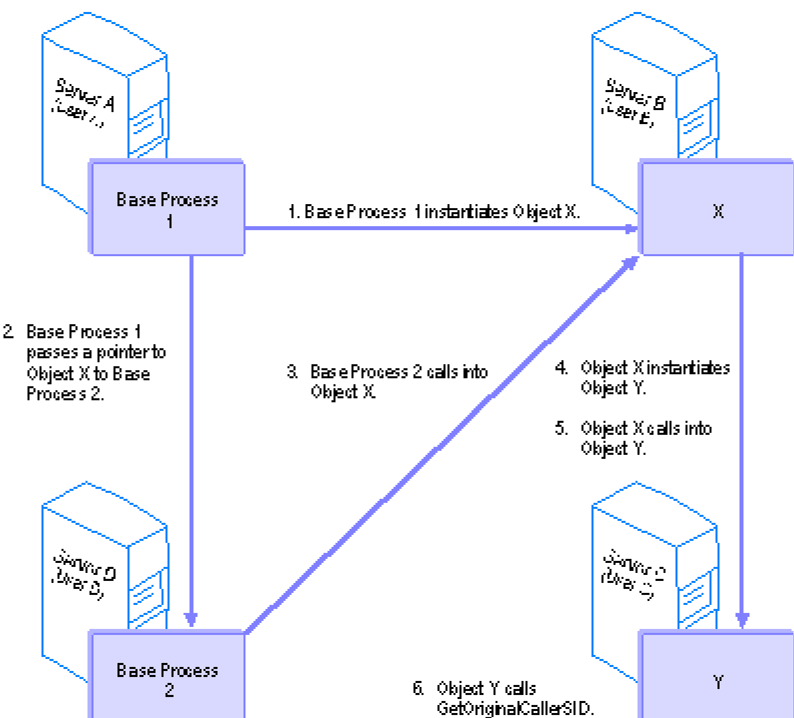
Note Visual Basic programmers can use the **SecurityProperty** object. The methods for **SecurityProperty** return user name strings instead of security identifiers (SIDs).

Security Identifiers (SIDs)

A Windows NT security identifier (SID) is a unique value that identifies a user or group. You can use SIDs to determine the exact identity of a user. Because of their uniqueness, SIDs do not have the flexibility of roles.

Callers and Creators

The following figure shows which SIDs are returned by the various methods on **ISecurityProperty** after a certain sequence of method calls.

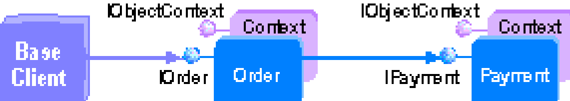


Calling the following methods on Object Y returns SIDs associated with these users:

- **GetDirectCallerSID** returns the SID associated with User B.
- **GetDirectCreatorSID** returns the SID associated with User B.
- **GetOriginalCallerSID** returns the SID associated with User D.
- **GetOriginalCreatorSID** returns the SID associated with User A.

Impersonation

Impersonation allows a thread to execute in a security context different from that of the process that owns the thread. Consider the following application scenario.



Basic Security Methods described an order-entry scenario in which the base client represents an employee submitting an order. In this scenario, the client is not authorized to use the Payment object and its associated database directly.

Suppose the base client were a report writer for an accounting program. In this case, you want to allow access to the Payment object's database. One way to accomplish this is for the Order object to impersonate the base client, allowing the database to use its own security checking to determine access privileges.

MTS does not promote the use of impersonation, but encourages role-based security. Security is simplified by the single-level of authorization provided by MTS, whereas the impersonation model has an *n*-level authorization architecture. The report-writer scenario can be simplified by defining a role, such as Accountant, to allow access to the database.

Error Handling

Fault Isolation and Failfast

MTS performs extensive internal integrity and consistency checks. If MTS encounters an unexpected internal error condition, it immediately terminates the process. This policy, called failfast, facilitates fault containment and results in more reliable and robust systems.

Consider a case in which MTS detects that one of its data structures is in a corrupted state. At this point, both the cause and the magnitude of the corruption are unknown. Unfortunately, MTS cannot tell how far the damage has spread. Certainly MTS is in an indeterminate state. But it does not run in isolation. Like other DLLs, it is hosted in a process environment and shares a single address space with the main program executable and many other DLLs. Consequently, it is safe to assume that the entire process has been corrupted. The process is immediately terminated to prevent it from spreading potentially corrupted information to other processes or, worse yet, from allowing corrupted data to be committed and made durable.

As a developer or administrator, you should inspect the Windows NT Event Viewer Application Log for details on any failfast or serious application errors.

Exceptions in MTS Objects

MTS does not allow exceptions to propagate outside of a context. If an exception occurs while executing within an MTS context and the application doesn't catch the exception before returning from the context, MTS catches the exception and terminates the process. Using the failfast policy in this case is based on the assumption that the exception has put the process into an indeterminate state—it is not safe to continue processing.

MTS Object Method Error Return Codes

MTS never changes the value of an HRESULT error code, such as E_UNEXPECTED or E_FAIL, returned by an MTS object method.

When an MTS object returns an HRESULT status code, such as S_OK or S_FALSE, MTS may convert the status code into an MTS error code before it returns to the caller. This occurs, for example, when the application returns S_OK after calling **SetComplete**; if the object is the root of an automatic transaction that fails to commit, the HRESULT is converted to CONTEXT_E_ABORTED.

When MTS converts a status code to an error code, it clears all of the method's output parameters. Returned references are released and the values of the returned object pointers are set to NULL.

See Also

MTS Error Diagnosis, MTS Error Codes

Developing Applications for MTS

Building MTS Applications

Explains the key concepts that a component application developer needs to understand for developing Microsoft Transaction Server (MTS) applications.

Creating a Simple ActiveX Component

Demonstrates how to create a component and register the component in the Microsoft Transaction Server run-time environment.

Building Scalable Components

Demonstrates how to use just-in-time activation to use server resources efficiently, resulting in more scalable applications and improved performance.

Building Transactional Components

Introduces transactional components and the benefits of running components within the same transaction.

Sharing State

Demonstrates how to use the Shared Property Manager to share state among multiple Transaction Server objects running in the same process.

Stateful Components

Discusses stateful components and outlines some of the issues associated with writing stateful application components.

Multiple Transactions

Explains the benefits of distributing work among multiple transactions.

Secured Components

Shows how to use Microsoft Transaction Server's security features to restrict the use of application features to designated users.

Building MTS Applications

This topic contains information that a component application developer needs to understand before building Microsoft Transaction Server (MTS) applications.

MTS Component Requirements

Business Logic in MTS Components

Packaging MTS Components

Calling MTS Components

Holding State in Objects

Database Access Interfaces with MTS

Developing MTS Components with Java

Debugging MTS Components

Automating MTS Deployment

MTS Error Diagnosis

MTS Component Requirements

An MTS component is a type of COM component that executes in the MTS run-time environment. In addition to the COM requirements, MTS requires that the component must be a dynamic-link library (DLL). Components that are implemented as executable files (.exe files) cannot execute in the MTS run-time environment. For example, if you build a Remote Automation server executable file with Microsoft Visual Basic, you must rebuild it as a DLL.

Additional Requirements for Visual C++ Components

- The component must have a standard class factory.

  The component DLL must implement and export the standard **DllGetClassObject** function and support the **IClassFactory** interface. MTS uses this interface to create objects. **IClassFactory::CreateInstance** must return a unique instance of an MTS object.
- The component must only export interfaces that use standard marshaling. For more information, see Passing Parameters.
- All component interfaces and coclasses must be described by a type library. The information in the type library is used by the MTS Explorer to extract information about the installed components.
- For custom interfaces that cannot be marshaled using standard Automation support, you must build the proxy-stub DLL with MIDL version 3.00.44 or later (provided with in the Microsoft Platform SDK for Windows NT version 4.0); use the –Oicf compiler switch; and link the DLL with the mtxih.lib library provided by MTS. The mtxih.lib library must be the first file that you link into your proxy-stub DLL. If the component has both a type library and a proxy-stub DLL, MTS will use the proxy-stub DLL.
- The component must export the **DllRegisterServer** function and perform self-registration of its CLSID, ProgID, interfaces, and type library in this routine.

Development tools such as Visual Basic and the ActiveX™ Template Library, which is available with Microsoft Visual C++, allow you to generate interfaces that COM can marshal automatically. These interfaces, known as dual interfaces, are derived from **IDispatch** and use the built-in Automation marshaling support.

Registering MTS Components

You manage MTS components by using the MTS Explorer. Before a component can run with context in the MTS run-time environment, you must use the MTS Explorer to define the component in the MTS catalog. In addition to keeping track of a component's basic COM attributes, such as the name of the implementation DLL, the MTS catalog maintains a set of MTS–specific attributes. MTS uses these attributes to provide capabilities in addition to those provided by COM. For example, the transaction attribute controls the transactional characteristics of a component.

The MTS Explorer assigns components to a package that controls the assignment of components to server processes and control client access to components.

Note MTS allows only a single server process associated with a given package to run on a computer at a time. MTS writes a warning event message to the log if you attempt to start a second instance of an already active package. However, COM does not explicitly disallow multiple servers running the same COM classes. MTS writes a warning message to the log in the event that two threads try to start the package at the same time. This event is especially likely on a symmetric multiprocessing (SMP) computer where the two package invocations are concurrent. In these cases, MTS enforces a rule of one server process for each package by terminating one of the extra packages. COM then connects to the one server process still running and successfully returns.

Running COM Components Under MTS

Exercise caution when registering a standard COM component (one developed without regard to MTS) to execute under MTS control.

First, ensure that references are safely passed between contexts (see Passing Object References).

Second, if the component uses other components, consider running them under MTS. Rewrite the code for creating objects in these components to use **CreateInstance** (see Creating MTS Objects).

Third, you can effectively use automatic transactions only with components that indicate the completion of their work by calling either the **SetComplete** or **SetAbort** methods. If a component does not use these methods, an automatic transaction can only be completed when the client releases the object. MTS will attempt to commit the transaction, but there is no way for the client to determine whether the transaction has been committed or aborted. Therefore, it is recommended that you do not register components as **Requires a transaction** or **Requires a new transaction** unless they use **SetComplete** and **SetAbort**.

Including Multiple Components in DLLs

You can implement multiple components in the same DLL. The MTS Explorer allows components from the same DLL to be installed in separate packages.

Including Type Libraries and Proxy-Stub DLLs in MTS Components

Development tools supporting ActiveX components can merge your type library or proxy-stub DLL with your implementation DLL. If you do not want to distribute your implementation DLL to client computers, keep your type libraries and proxy-stub DLLs separate from your implementation DLLs. The client only needs a type library or custom proxy-stub DLL to use your server application remotely.

Business Logic in MTS Components

This topic describes how to enact business logic in MTS components.

Granularity is determined by the number of tasks performed by a component. The granularity of a component affects the performance, debugging, and reusability of your MTS components. A *fine-grained* component performs a single task, such as calculating tax on a sales order. Fine-grained components consume and release resources quickly after completing a task. A component that enacts a single business rule can facilitate testing packages, because isolating individual tasks in components makes testing your applications easier. In addition, fine-grained components are easily reused in other packages. In the following example, a component performs a single task: adding a customer record to the database.

```
Function Update(ByVal strEmail As String, _
ByVal bNewCust As Boolean, ByVal strContact As String,_
ByVal strPhoneNumber As String, _
ByVal strNightPhoneNumber As String)

    Dim ctxObject As ObjectContext
    Set ctxObject = GetObjectContext

    On Error GoTo ErrorHandler

' Code accesses the customer row from the database.
' Customer information is updated with information
' that was passed in.
'
    ctxObject.SetComplete

    Exit Function
```

This simple component uses system resources efficiently (passing parameters by value), is easy to debug (single function), and also reusable in any other application that maintains customer data.

A *coarse-grained* component performs multiple tasks. Coarse-grained components are generally harder to debug and reuse in applications. For example, a PlaceOrder component might add a new order, update inventory, and update customer information. PlaceOrder is a more coarsely grained component because it performs more "work" by adding, updating, and deleting customer, inventory and order information.

For more information about components' shared resources, see Holding State in Objects.

Packaging MTS Components

This document describes how you should package your MTS components. Consider the following design issues when defining package boundaries:

- Activation
- Shared resources
- Fault isolation
- Security isolation

Activation

You can select either of the following Activation levels for your packages:

- *Library* (running within the same process as the client that creates the object)
- *Server* (on the same computer but in a different process)

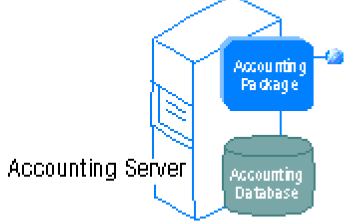
MTS provides a way to set up remote components by using the Remote Computer and Remote Component folders in the MTS Explorer hierarchy. For more information about "pulling" or "pushing" components between computers, see the *Administrator's Guide*.

By default, components run in a server process on the local computer. If you run your components within the MTS server process, you enable resource sharing, security, and easier administration by using the MTS Explorer for your component. Running components in-process provides an immediate performance benefit, because you do not have to marshal parameters cross-process. However, in-process components do not support declarative security and you lose fault isolation.

Sidebar: In-process Components and Security

Note that in-process components do not support declarative security or offer the benefits of process isolation. In-process components will run in any process that creates the component. Role checking is disabled between in-process components because **IsCallerInRole** returns True. In other words, the direct caller always passes the authorization check.

Also, it is recommended that you place your components as close as possible to the data source. If you are building a distributed application with a number of packages running on local and remote servers, try to group your components according to the location of your data. For example, in the following figure, the Accounting server hosts an Accounting package and Accounting database.



Shared Resources

Sharing resources in a multiuser environment results in faster applications that scale more easily. Note that only components marked with the **Local** activation setting can share resources. Package your components to take advantage of the resource sharing and pooling that MTS provides for your application.

Pool your resources by server process.Note that MTS runs each hosted package in a separate server process. The fewer pools you have running on your server, the more efficiently you pool resources, so try to group components that share "expensive" resources, such as connections to a specific database. If you reuse the expensive resources within your package, you can greatly improve the

performance and scaling of your application. For example, if you have a database lookup and a database update component running in a customer maintenance application, package those components together so that they can share database connections.

Fault Isolation

Fault isolation requires separating components into packages that can operate in their own server process. Components in the same package share the same server process if all the activation settings are the same. By placing components in separate packages, you can mitigate the impact of a component failure because each package runs in a separate server process.

You can also use fault isolation to test new components. You can stage updates to MTS applications by introducing new components. Fault isolation for packages greatly reduces the risk of your local server package failing when you introduce a new component to a shared environment.

Security Isolation

MTS *security roles* represent a logical group of user accounts which are mapped to Microsoft Windows NT® domain users and groups during the deployment of the package. You can use the MTS Explorer to define *declarative authorization checking* by applying roles to components and component interfaces. Applying a security role to a component defines access privileges for any user assigned as a member of that security role. Users not assigned to a role with access privileges to a package cannot use the package. Because security authorization occurs between packages rather than between components within a package, it is recommended that you consider the MTS security model when determining your package boundaries. Note that security isolation only applies to packages with components running under the **Server** activation setting.

Security authorization is checked when a method call crosses a package boundary, such as when a client calls into a package or one package calls another. When you package your components, make sure you group components that can safely call each other without requiring security checks within one package.

All components within a package run under the identity established for the package. If you run under different identities, separate them into two different packages.

You can use declarative security between the client and server, and database security based on package identity between the server and data source. You can restrict access to a data source by assigning an identity to a package, and configuring the database to accept updates according to package identity.

If you use package identity to set up your database security, the database recognizes the package identity as a single user. If database access occurs under an identity set by the package, the database connection set up for the package identity name can be used by all the users mapped to role or roles for that package. This kind of resource sharing improves application performance and scalability.

Calling MTS Components

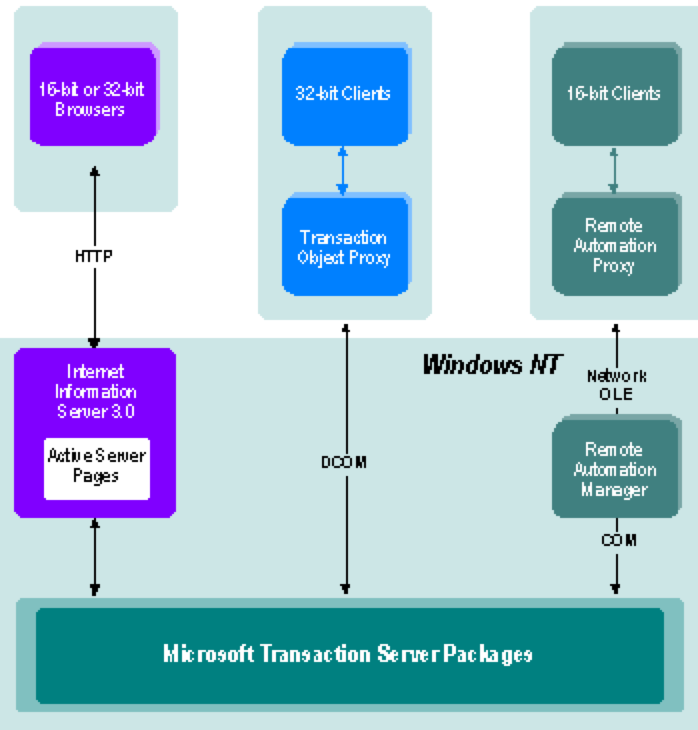
This topic covers the following:

- Calling MTS Components using DCOM
- Calling MTS Components from an Active Server Page
- Calling MTS Components from a Web Browser-Resident Component

For more information on the methods for creating MTS objects, see Creating MTS Objects.

Calling MTS Components from a Client Application

MTS components can be located on a separate computer from the client. A client can call a remote MTS component using DCOM, HTTP, or Remote Automation. To run an MTS component on the client computer, the client computer must have MTS installed.



Calling MTS Components through DCOM

DCOM is the standard transport for calling MTS components. To enable DCOM calls to MTS components, you must configure the following:

Client Registry Settings The easiest way to configure your client application to call a remote MTS component is to use the application executable utility, which automatically configures client registry settings. For more information,see the *Administrator's Guide*.

DCOM Security Settings You may have to configure the Impersonation Level and Authentication Level on both client and server computers. MTS works properly using the default values for these settings: **Identify** for Impersonation Level and **Connect** for Authentication Level. Make the necessary changes in the MTS Explorer at the package level. Changing default settings by using the DCOM configuration utility (dcomcnfg.exe) is not recommended.

If you want to use Microsoft Windows 95 clients with MTS, install DCOM for Windows® 95. For the latest information on DCOM support for Windows 95, see http://www.microsoft.com/oledev on the World Wide Web.

Calling MTS Components through Remote Automation

Remote Automation was introduced with Visual Basic version 4.0, before the introduction of DCOM. It is useful for 16-bit clients, because DCOM works only in 32-bit environments. To use Remote Automation with MTS, the Remote Automation Manager (RACMAN) must be running on the server where the MTS components are installed. For more information, see the Visual Basic documentation.

Note You cannot use MTS security Remote Automation since all calls are made using the RACMAN identity. Because RACMAN does not impersonate when calling the components on the server, the client identity cannot be determined.

Calling MTS Components through HTTP

There are two ways a client can call an MTS component through HTTP:

- Call an Active Server Page (ASP), which in turn calls the MTS component using DCOM.
- Call the MTS component from a Web browser – resident component using the ActiveX Data Objects (ADO) Remote Data Service (RDS), which in turn uses HTTP. For more information about RDS, see http://www.microsoft.com/adc.

Calling MTS Components from an Active Server Page

You can call MTS components from Active Server Pages (ASPs). You can create an MTS object from an ASP by calling **Server.CreateObject**. Note that if the MTS component has implemented the **OnStartPage** and **OnEndPage** methods, the **OnStartPage** method is called at this time.

You can run your MTS components in-process with or out-of-process with Internet Information Server (IIS). If you run your MTS components in-process with IIS, be aware that if MTS encounters an unexpected internal error condition or an unhandled application error such as a general-protection fault inside a component method call, it immediately results in a failfast, thus terminating the process and IIS.

By default, IIS 3.0 disables calling out-of-process components. To enable calling out-of-process components, modify the following registry entry

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\ASP\Parameters

by setting the **AllowOutOfProcCmpnts** key to 1.

Calling MTS Components from a Web Browser-Resident Component

You can call an MTS component from a Web browser – resident component. Use the application executable utility to configure that client, and then use the HTML <OBJECT> tag to call that component. You can also use the <OBJECT> tag to create an MTS object in-process with the browser client. Remember that MTS must be installed on the client computer for an MTS component to run in-process.

The component should be made safe for scripting, either through a component category entry in the registry, or by supporting the **IObjectSafety** interface.

Remote Data Service (RDS) also allows you to create web browser – resident components using the <OBJECT> tag. RDS supports the following:

- HTTP
- HTTPS (HTTP over Secure Socket Layer)
- DCOM
- In-process server

Except for in-process objects, the **CreateObject** method of the **DataSpace** object creates a proxy for the MTS object that runs in a local or remote server process.

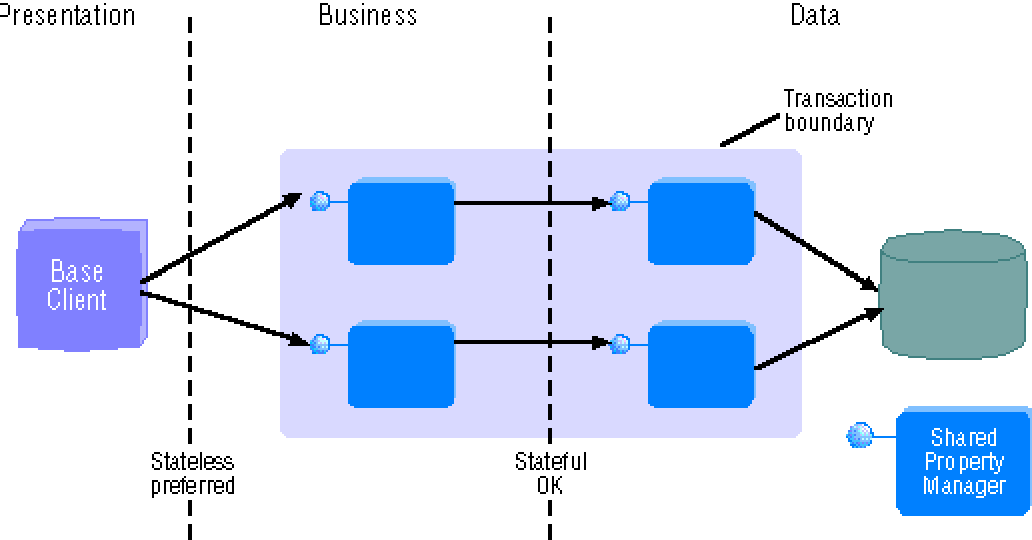
You must configure the following registry key to the Prog ID of the object that you want to call:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\ADCLaunch

Holding State in Objects

Although there are many benefits to using *stateless* MTS objects, there are cases where holding state is desirable. This topic provides some guidelines in deciding where state is held in your application.

The following diagram shows a *three-tier architecture*:



Typically, the latency between tiers differs greatly. Calls between the *presentation tier* and *business tier* are often an order of magnitude slower than calls between the business tier and *data tier*. As a result, held state is more costly when calling into the business tier.

However, it often makes sense to hold state within the *transaction boundary* itself. For example, the objects in the data tier may represent a complex join across many tables in separate databases. Reconstructing the data object state is potentially more inefficient than the cost of the resources held by those objects while they remain active.

Since objects lose state on transaction boundaries, if you need to hold state across transactions, use the Shared Property Manager or store the state in a database.

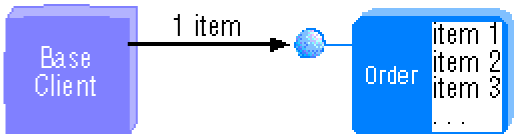
Example: Order-Entry Application

There are two separate issues when considering the effects of holding state in an application:

- Network roundtrips More frequent network roundtrips and slower connections extend the *lifetime* of the called MTS object.
- Held resources Holding state often means holding onto a resource, such as a database connection, and potentially, locks on the database.

Consider the example of an online shopping application. The client chooses items from a catalog and submits an order. Order processing is handled by a business object, which in turn stores the order in a database (not shown).

One way of building the application is for the client to call an **Order** object, with each call adding or removing an item from the order:

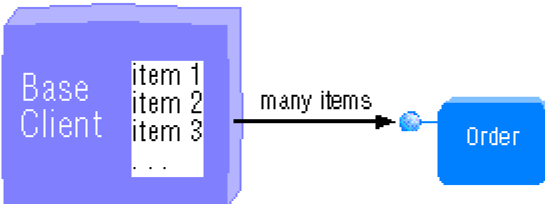


This application has the following properties:

- Client maintains no state.

- Server maintains state across multiple calls.
- Many network roundtrips.
- High contention for resources. The database connection is held for the lifetime of the **Order** object. This is not a very scalable solution.

You can require that the client cache the items in an array or recordset:



This application has the following properties:

- Stateful client.
- Server is virtually stateless with one call to the server.
- Fewer network roundtrips.
- Less contention for resources. This is a scalable solution.

Concurrency

In addition to network bandwidth and resources, concurrency affects application performance. There are two types of *concurrency*:

- **Pessimistic** As soon as editing begins, the database locks the records being changed. The records are unlocked when all changes are complete. No two users can access the same record at the same time.
- **Optimistic** The database locks the records being changed only when the changes are committed. Two users can access the same record at the same time, and the database must be able to reconcile, or simply reject, changed records that have been edited by multiple users prior to commit.

Implementing a server cache implies *optimistic concurrency*. The server does not have to hold locks on the database, thus freeing resources.

However, if there is high contention for the resource, *pessimistic concurrency* may be preferred. It is easier to reject a request to access a database and have the server try again than it is to reconcile cached, out-of-date data with a rapidly changing database.

Database Access Interfaces with MTS

This topic describes the database access interface options for MTS applications. You can use the Open Database Connectivity (ODBC) Application Programming Interface (API) to access a *resource manager* (which is a system service that manages durable data), or a data access model that functions over the ODBC layer. Because the ODBC version 3.0 Driver Manager is an MTS resource dispenser, data accessed via ODBC is automatically protected by your object's transaction. For object transactions, an ODBC-compliant database must support the following:

- The database's ODBC driver must be thread safe. It also must be able to connect to the driver from one thread, use the connection from another thread, and disconnect from another thread.
- If ODBC is used from within a transactional component, then the ODBC driver must also support the SQL_ATTR_ENLIST_IN_DTC connection attribute. This is how the ODBC Driver Manager asks the ODBC driver to enlist a connection on a transaction. You can make your component transactional by setting the transaction property for your component in the MTS Explorer. If you are using a database without a resource dispenser that can recognize MTS transactions, contact your database vendor to obtain the required support.

The following table summarizes database requirements for full MTS support.

Requirements	Description	Resources (if applicable)
Support for the OLE transactions specification, or support for XA protocol	Enables direct interaction with Distributed Transaction Coordinator (DTC). Use the XA Mapper to interact with DTC	MTS Beta SDK
ODBC driver	Platform requirement for MTS server components	ODBC version 3.0 SDK
ODBC driver support for the ODBC version 3.0 SetConnectAttr SQL_ATTR_ENLIST_ IN_DTC call.	MTS uses this call to pass the transaction identifier to the ODBC driver. The ODBC driver then passes the transaction identifier to the database engine.	ODBC version 3.0 SDK
Fully thread-safe ODBC driver	ODBC driver must be able to handle concurrent calls from any thread at any time.	ODBC version 3.0 SDK
ODBC driver must not require thread affinity	ODBC driver must be able to connect to the driver from one thread, use the connection from another thread, and disconnect from another thread.	ODBC version 3.0 SDK

If a memory access violation in the mtx.exe process occurs within the driver after 60 seconds of inactivity, you may be using an ODBC driver that is not thread safe or requires thread affinity. The fault occurs in the driver when the inactive connections are being disconnected.

MTS Distributed Transaction Coordinator

MTS uses the services of the Microsoft Distributed Transaction Coordinator (DTC) for transaction coordination. DTC is a system service that coordinates transactions that span multiple resource managers. Work can be committed as a single transaction, even if it spans multiple resource managers, potentially on separate computers. DTC was initially released as part of Microsoft SQL Server version 6.5, and is included as part of MTS. DTC implements a *two-phase commit protocol*

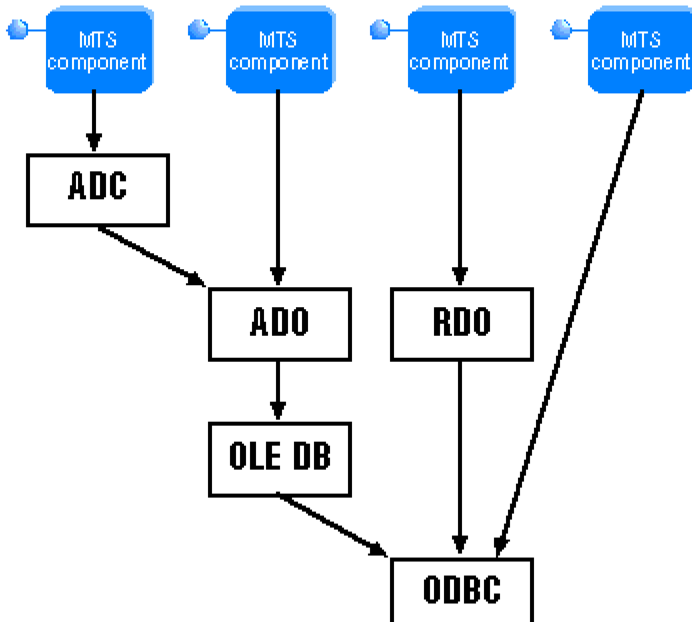
that ensures that the transaction outcome (either *commit* or *abort*) is consistent across all resource managers involved in a transaction. DTC supports resource managers that implement OLE Transactions, X/Open XA protocols, and LU 6.2 Sync Level 2.

Choosing your Data Access Model

The following table summarizes commonly used data access models supported by MTS.

Interface	Description
Microsoft ActiveX Data Objects (ADO), Remote Data Service (RDS)	ADO offers one common yet extensible programming model for accessing data. ADO includes the ability to pass query results (*Recordsets*) between server and client, and the ability to pass updated Recordsets from client to server using RDS.
OLE DB	OLE DB is a low-level interface that provides uniform access to any tabular data source. You cannot call OLE DB interfaces directly from Microsoft® Visual Basic® because OLE DB is a pointer-based interface. A Visual Basic client can access an OLE DB data source through ADO.
Open DataBase Connectivity (ODBC)	ODBC is a recognized standard interface to relational data sources. ODBC is fast and provides a universal interface that is not optimized for any specific data source.
Remote Data Objects (RDO)	RDO is a thin object layer interface to the ODBC API. It is specifically designed to access remote ODBC relational data sources.

The diagram below illustrates how MTS components interact with the different data access interfaces:



ADO is not specifically designed for relational or ISAM databases, but as an object interface to *any* data source. ADO can access relational databases, ISAM, text, hierarchical, or any type of data source, as long as a data access provider exists. ADO is built around a set of core functions that all data sources are expected to implement. ADO can access native OLE DB data sources, including a specific OLE DB provider that provides access to ODBC drivers. ADO ships with the OLE DB Software Development Kit (SDK).

RDO does have some functionality that is not currently implemented in ADO, including the following:

- Events on the Engine, Connection, Resultset, and Column objects
- Asynchronous operations
- Queries as methods

- Enhanced batch-mode error and contingency handling
- Tight integration with Visual Basic, as in the Query Connection designer and TSQL debugger.

Future versions of ADO will provide a superset of RDO version 2.0 functionality and provide a far more sophisticated interface, in addition to an easier programming model. Because ADO is an Automation-based component, any application or language capable of working with Automation objects can use it.

Developing MTS Components with Java

You can develop Java MTS components using tools provided by MTS and Visual J++. It is also recommended that you install the latest version of the Microsoft SDK for Java, available at http://www.microsoft.com/java.

This section contains the following topics:

- Implementing a Component in Java
- Using an MTS Component from Java
- Using the Java Sample Bank Components

Implementing a Component in Java

To implement a component in Java, follow these steps

1. Run the ActiveX Component Wizard for Java (available with Visual J++) for each Java class file. Use the wizard to create new IDL files.
2. Modify the IDL files to add JAVACLASS and PROGID to the coclass attributes. See "Using IDL Files with Java Components."
3. Run the ActiveX Component Wizard for Java again. Use the IDL files that you created in Step 1 to create type libraries for your components.

   This will create a set of Java class files, typically under \%systemroot%\Java\Trustlib. It will create one class file for each custom interface, and one class file for each coclass in the library.
4. Run JAVAGUID against each class file generated in Step 3. See "Working with GUIDs in Java" for more information.
5. Recompile your Java implementation classes.
6. Run EXEGEN to convert the type libraries and class files into a DLL. See "Using EXEGEN to Create DLLs."
7. Use the MTS Explorer to install the DLL.

Using IDL Files with Java Components

To specify the custom attributes in a type library, add the following in your IDL or ODL file:

```
#include <JavaAttr.h>
```

Within the attributes section of a coclass, specify the JAVACLASS:

```
JAVACLASS("package.class")
```

You may optionally specify a PROGID:

```
PROGID("Progid")
```

For example:

```
[
   uuid(a2cda060-2d38-11d0-b94b-0080c7394688),
   helpstring("Account Class"),
   JAVACLASS("Account.AccountObj"),
   PROGID("Bank.Account.VJ"),
   TRANSACTION_REQUIRED
]
coclass CAccount
{
   [default] interface IAccount;
};
```

Using EXEGEN to Create DLLs

EXEGEN is the Java executable file generator. To use this file, copy it to the appropriate destination folders (usually \JavaSDK\bin). This version of EXEGEN.EXE is capable of creating DLL files from Java classes, and can also include user-specified resources in its output files. This version of EXEGEN no longer supports the /base: directive. Class files are always included with the proper name. It supports a new /D directive that causes it to generate a DLL file instead of an EXE.

EXEGEN is now capable of reading five types of input files:

- Java class files
- RES files containg resources to be included
- Executable files containg resources to be included
- TLB files containing type libraries to be included
- Text files describing which classes should be registered (DLL only).

If you use EXEGEN to create a DLL, the DLL can self-register any included Java classes that implement COM objects. There are two ways to tell EXEGEN which classes should be registered:

- Include a type library that contains custom attributes for the classes. This is the preferred method.
  – or –
- Include a text file as input that gives EXEGEN the necessary directions. Each line of the text file describes one Java class, using the following keywords:
  - class:JavaClassName
    Required keyword.
  - clsid:{.....}
    Optional keyword that specifies the clsid GUID. If omitted, EXEGEN chooses a unique GUID.
  - progid:ProgId
    Optional keyword. If omitted, the class will be registered without a progid.

Working with GUIDs in Java

When an MTS method uses a GUID parameter, you must pass an instance of class com.ms.com._Guid. Do not use class Guid, CLSID or IID from package com.ms.com; they will not work and they are deprecated. The definition of class _Guid is:

```
package com.ms.com;
public final class _Guid {

   // Constructors
   public _Guid (String s);
   public _Guid (byte[] b);
   public _Guid (int a, short b, short c,
      byte b0, byte b1, byte b2, byte b3,
      byte b4, byte b5, byte b6, byte b7);
   public _Guid ();

   // methods
   public void set(byte[] b);
   public void set(String s);
   public void set(int a, short b, short c,
      byte b0, byte b1, byte b2, byte b3,
      byte b4, byte b5, byte b6, byte b7);

   public byte[] toByteArray();
   public String toString();
```

```
}
```

Instances of this class can be constructed from a String (in the form "{00000000-0000-0000-0000-000000000000}"), from an array of 16 bytes, or from the usual parts of a Guid. Once constructed, the value can also be changed. Method toByteArray will return an array of 16 bytes as stored in the Guid, and method toString will return a string in the same form used by the constructor.

JAVAGUID.EXE

Microsoft Transaction Server supplies a tool, JAVAGUID.EXE, that will post-process the output of JAVATLB. The following occurs for each class file:

- If any method takes a GUID as a parameter, the class of that parameter will be changed to com.ms.com._Guid.
- If the class file is an interface derived from a type library, a public static final member named **iid** will be added to the class. This member will contain the interface ID of the interface.
- If the class file represents a coclass derived from a type library, a public static final member named **clsid** will be added to the class. This member will contain the CLSID of the class.

The **clsid** and **iid** members that JAVAGUID adds are useful as parameters to **IObjectContext.CreateInstance** and **ITransactionContextEx.CreateInstance**.

JAVAGUID can only be executed from the command line. It takes one or more parameters which are names of class files to update.

JAVATLB will eventually be updated to make JAVAGUID unnecessary.

Using an MTS Component from Java

To use an MTS component from Java, run the Java Type Library Wizard against the type library for the component. This will create several Java class files, typically under \%systemroot%\Java\TrustLib. It will create one class file for each custom interface, and one class file for each coclass in the library.

Assume, for example, that the type library contained one interface named IMyInterface, and one coclass, named CMyClass.

From Java, you can create a new instance of the component by executing

```
new CMyClass()
```

If you want to control transaction boundaries in the class, you can execute

```
ITransactionContextEx.CreateInstance ( CMyClass.clsid, IMyInterface.iid )
```

You should never call Java's **new** operator on the class that you implemented. Instead, use one of the following techniques:

- Use Java's new operator on the class created by the Java Type Library Wizard. This will cause the Java VM to call **CoCreateInstance**.
- Call **MTx.GetObjectContext().CreateInstance** (*clsid*, *iid*);
  This will create a new instance in the same activity as the current instance. This only works if the calling code is itself an MTS component.
- If you have a reference to an **ITransactionContextEx** object, call its **CreateInstance** method. This will create a new instance in the transaction owned by the **ITransactionContextEx** object.

All of these techinques will result in the creation of a new instance of the class that you implemented.

Using the Java Sample Bank Components

The Java Sample Bank components are automatically configured by MTS Setup and require no additional steps in order to run them.

If you want to recompile the Java Sample Bank components, follow these steps:

1. Run the SetJavaDev.bat file located in the \mts\Samples\Account.VJ folder. Javatlb.exe must be in your path for this batch file to run properly.
2. Recompile your Java component implementation classes.
3. After you recompile the component classes, use the mkdll.bat file located in the \mts\Samples\Account.VJ folder to generate and register vjacct.dll. Exegen.exe must be in your path for this batch file to run properly. You can also add running mkdll.bat as a build step to your Visual J++ project to simplify recompiling.
4. Using the MTS Explorer, import the new components into the Sample Bank package.

Debugging MTS Components

This document describes techniques for debugging MTS components written in Microsoft® Visual Basic®, Microsoft Visual C++®, and Microsoft Visual J++™. These techniques are just suggestions for successfully debugging MTS components; you can choose your debugging environment and techniques according to your application needs.

If you are using MTS components in a distributed environment, it is recommended that you debug your components on a single computer before deploying to multiple servers. Components that function without error in a package on a local computer usually run successfully over a distributed network. If you do encounter problems with distributed components, you must test and debug both the client and server machines to determine the problem. It is also recommended that you stress test your application with as many clients as possible. You can build a test client that simulates multiple clients to perform the stress test on your application.

For debugging MTS components written in a specific language, see the following topics:

Debugging Visual Basic MTS Components

Debugging Visual C++ MTS Components

Debugging Java Classes

Debugging Visual Basic MTS Components

Microsoft Transaction Server components written in Visual Basic version 5.0 or Visual C++ version 5.0 can be debugged in the Microsoft Visual Studio 97 Integrated Development Environment (IDE).

If you want to debug your components after they are compiled, you cannot use the Visual Basic 5.0 debugger, which only debugs at design time. To debug a compiled Visual Basic component, you will need to use the functionality of the Visual Studio 97 debugger.

Follow these steps to configure Visual Studio to debug MTS components built with Visual Basic 5.0:

1. In Visual Basic, click **Properties** on the **Project** menu and then click the **Compile** tab to select the **Compile to Native Code** and the **Create Symbolic Debug Info** checkbox. It is also recommended that you select the **No Optimization** checkbox while debugging.
2. In the MTS Explorer, right-click the package in which your component is installed, and select the **Properties** option. Place your cursor over the Package ID, and select and copy the GUID to the clipboard.
3. Open Visual Studio. On the **File** menu, click **Open** and select the DLL containing the component that you want to debug.
4. Select **Project Settings**, and then click the **Debug** tab. Select the MTS executable for the debug session (\mtx\mtx.exe). Enter the program arguments as /p:{<package GUID>} for the package GUID that you copied from the package properties. MTS 2.0 allows for the package name to be used in place of the GUID. Open the .cls files containing the code that you want to debug and then set your breakpoints. If you also want to display variable information in the debug environment, go to the Visual Studio Tools menu, select **Options**, and then select the **Debug** tab. In the **Debug** tab, place a check next to **Display Unicode strings**.
5. In the MTS Explorer, shut down all server processes.
6. In Visual Studio, select **Build**, then select **Start Debug**. Then select **Go** to run the server process that will host your component(s), and set breakpoints to step through your code.
7. Run your client application to access and debug your components in Visual Studio.
8. Before you deploy your application, remember to select one of the optimizing options in the **Compile** tab on the **Project** menu of Visual Basic (set to **No Optimization** in Step 1), clear the **Create Symbolic Debug Info** checkbox, and recompile the project.

To facilitate application debugging using Visual Basic 5.0, a component that uses **ObjectContext** can be debugged by enabling a special version of the object context. This debug-only version is enabled by creating the registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Transaction Server\Debug\RunWithoutContext

Note that when running in debug mode, none of the functionality of MTS is enabled. **GetObjectContext** will return the debug **ObjectContext** rather than returning **Nothing**.

When running in this debug mode, the **ObjectContext** operates as follows:

- **ObjectContext.CreateInstance** - calls COM **CoCreateInstance** (no context flows, no transactions, and so on)
- **ObjectContext.SetComplete** - no effect
- **ObjectContext.SetAbort** - no effect
- **ObjectContext.EnableCommit** - no effect
- **ObjectContext.DisableCommit** - no effect
- **ObjectContext.IsInTransaction** - returns FALSE
- **ObjectContext.IsSecurityEnabled** - returns FALSE
- **ObjectContext.IsCallerInRole** - returns TRUE (same as normal when **IsSecurityEnabled** is FALSE)

You can also develop your own testing message box functions to generate an *assert* in an MTS Visual Basic component. The following sample code can to used to display error messages while debugging Visual Basic code. You can also use this in conjunction with the Microsoft Windows NT® debugger (WinDbg.exe), which is a 32-bit application that, along with a collection of DLLs, is used for debugging the Kernel, device drivers, and applications. Note that you must enter DEBUGGING = -1 in the **Conditional Compilation** dialog box (located on the **Make** tab of the **Project Properties** dialog box) to enable the assert.

The following code provides an example.

```
#If DEBUGGING Then
    'API Functions
    Private Declare Sub OutputDebugStringA _
        Lib "KERNEL32" (ByVal strError As String)
    Private Declare Function MessageBoxA _
        Lib "USER32" (ByVal hwnd As Long, _
        ByVal lpText As String, _
        ByVal lpCaption As String, _
        ByVal uType As Long) As Long
    'API Constants
    Private Const API_NULL                  As Long = 0
    Private Const MB_ICONERROR              As Long = &H10
    Private Const MB_SERVICE_NOTIFICATION As Long = &H200000

Public Sub DebugPrint(ByVal strError As String)
    Call OutputDebugStringA(strError)
End Sub

Public Sub DebugMessage(ByVal strError As String)
   Dim lngReturn As Long
        lngReturn = MessageBoxA(API_NULL, strError, "Error In Component", _
            MB_ICONERROR Or MB_SERVICE_NOTIFICATION)
End Sub
#End If
```

You can then run checks through your code to aid stress debugging, such as in the following code:

```
SetobjObjectContext=GetObjectContext()
#If DEBUGGING Then
If objObjectContext Is Nothing Then Call DebugMessage("Context is Not
Available")
#End If
```

Debugging Visual C++ MTS Components

You can use Visual Studio 97 to debug MTS components written in Visual C++, including components that call SQL Server functions or stored procedures. For more information, see Debugging Visual Basic MTS Components.

The following information applies to components that have their activation property set to **In a dedicated server process**.

Microsoft Transaction Server supports the COM transparent remote debugging infrastructure. If transparent remote debugging is enabled, then stepping into a client process will automatically stop at the actual object's code in the server process, even if the server is on a different computer on the network. A debugging session is automatically started on the server process if necessary. Similarly, single stepping past the return address of code in a server object will automatically stop just past the corresponding call site in the client's process.

In Microsoft Visual C++, selecting the **OLE RPC debugging** check box (on the **Tools** menu, select the **Options** submenu and choose the **Debug** property sheet) enables transparent remote debugging. It is not known at this time whether other debuggers support this infrastructure.

You can also debug your Microsoft Transaction Server component DLL in Visual C++ by performing the following steps. Each of these steps is made either inside the MTS Explorer or inside of a Visual C++ session with your MTS DLL project.

1 Shutdown server processes using the MTS Explorer. To do this, right-click **My Computer**, and select **Shutdown Server Process**.

3 In your Visual C++ session, under **Project**, **Settings**, **Debug**, **General**, set the program arguments to the following string: **"/p:** *PackageName***"**, for example:

```
/p: "Sample Bank"
```

4 In the same property sheet, set the executable to the full path of the Mtx.exe process, for example: **"c:\MTx\MTx.exe"**.

5 Set breakpoints in your component DLL, and you are ready to debug.

6 Run the server process (in the **Build** menu, select **Start Debug** and click **Go**.)

The following information applies to in-process component DLLs that have their activation property set to **In the creator's process**.

You can debug your in-process MTS component DLL in Visual C++ by performing the following steps. Each of these steps is made inside a Visual C++ session with your base process project.

1 Set the component DLL under **Build, Settings, Debug, Additional DLLs**.

2 Now you are ready to step into or set breakpoints in your component DLL at will.

If you are using Visual Studio and Microsoft Foundation Classes (MFC) to debug, the TRACE macro can facilitate your debugging. The TRACE macro is an output debug function that traces debugging output to evaluate argument validity. The TRACE macro expressions specify a variable number of arguments that are used in exactly the same way that a variable number of arguments are used in the run-time function **printf**. The TRACE macro provides similar functionality to the **printf** function by sending a formatted string to a dump device such as a file or debug monitor. Like **printf** for C programs under MS-DOS, the TRACE macro is a convenient way to track the value of variables as your program executes. In the Debug environment, the TRACE macro output goes to afxDump. In the Release environment, the TRACE macro output does nothing.

Example:

```
// example for TRACE
int i = 1;
char sz[] = "one";
TRACE( "Integer = %d, String = %s\n", i, sz );
```

```
// Output: 'Integer = 1, String = one'
```

The TRACE macro is available only in the debug version of MFC, but a similar function could be written for use without MFC. For more information on using the TRACE macro, see the "MFC Debugging Support" section in *Microsoft Visual C++ Programmer's Guide*.

Note that you should avoid using standard ASSERT code in Visual C++. Instead, it is recommended that you write assert macros like a **MessageBox** using the MB_SERVICE_NOTIFICATION flag, and TRACE macro statements using the **OutputDebugString** function call.

Debugging Java Classes

Debug your Java classes as thoroughly as possible before converting the Java classes into MTS components. Note that once your Java class is converted into an MTS component, it is not possible to step through the code in the Visual J++ debugger, or in any current debugging tool as well.

Sidebar: Using Visual J++ to Debug Java Classes

Microsoft Visual J++ (VJ++) provides a Java debugger that you can use to set breakpoints in your code. Note that when you are using VJ++ to debug, if you set a breakpoint in a Java source file before starting the debugging session, Visual J++ may not stop on the breakpoint. For performance reasons, the debugger preloads only the *main class* of your project. The main class is either the class with the same name as the project or the class you specify in VJ++. If you use the editor to set breakpoints in other classes before the classes are loaded, the breakpoints are disabled.

You can choose one of the following options to load the correct class so that the debugger stops at breakpoints.

- Select the class in the category **Additional Classes**, located on the **Debug** tab of the **Project Settings** dialog box, and make sure the first column is checked. This loads the class when the debugging session starts.
- Right-click a method in the ClassView pane of the **Project Workspace** and select **Set Breakpoint** from the **Shortcut** menu. This causes a break when program execution enters the method.
- Set the breakpoint after Visual J++ has loaded the class during debugging. You may need to step through your Java source until the class is loaded.

When a method has one or more overloaded versions and shows up as a called method in the **Call Stack** window, the type and value for the parameters are not displayed in some cases. It appears as though the method takes no parameters. This occurs when the called method is not defined as the first version of the overloaded method in the class definition. For example, see the following class definition:

```
public class Test
    {
        int method(short s)
        {
            return s;
        }

        int method(int i)
        {
            return i;
        }
    }
```

If you were looking at a call to the second version of the method in the **Call Stack** window, it would appear without the type and value for the method:

```
method()
```

To view the method's parameters, change the order of the method overloads so that the method that you are currently debugging is first in the class definition.

printf-style Debugging

You can use **printf**-style debugging to debug your Java classes without using a debugger. printf-style debugging involves including status text messages into your code, allowing you to "step through" your code without a debugger. You can also use printf-style debugging to return error information. The following code shows how you can add a `System.out.println` call to the **try** clause of the

Hellojtx.HelloObj.SayHello sample.

```
try
{
System.out.println("This message is from the HelloObj implementation");
   result[0] = "Hello from simple MTS Java sample";
   MTx.GetObjectContext().SetComplete();
   return 0;
}
```

The client must be a Java client class, and you must use the JVIEW console window to run that class. Note that you need to configure your component to run in the process of its caller, which is in this case JVIEW. Otherwise, this debugging technique results in your component running in the MTS server process (mtx.exe), which would put the println output in the bit bucket rather than the JVIEW console window.

Use the MTS Explorer to configure your component to run in the caller's process by following these steps.

1 Right-click the component.

2 Click the **Properties** option.

3 Click the **Activation** tab and clear the **In a server process on this computer** checkbox.

4 Select the **In the creator's process**... checkbox.

5 Reload the **Client** class. Your component's `println` calls will be visible in the JVIEW console window.

Using the AWT Classes

You can also use the AWT (Abstract Window Toolkit) classes to display intermediate results, even if your component is running in a server process. The java.awt package provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields.

The following example demonstrates how to use the AWT classes to display intermediate results in a dialog box:

```
import java.awt.*;

public final class MyMessage extends Frame
{

   private Button closeButton;
   private Label textLabel;

   // constructor
   public MyMessage(String msg)
   {
      super("Debug Window");

      Panel panel;

      textLabel = new Label (msg, Label.CENTER);
      closeButton = new Button ("Close");

      setLayout (new BorderLayout (15, 15));
      add ("Center", textLabel);

      add ("South", closeButton);
```

```
      pack();
      show();
   }

   public boolean action (Event e, Object arg)
   {

      if (e.target == closeButton)
      {
         hide();
         dispose();
         return true;
      }

      return false;
   }

}
```

Asynchronous Java Garbage Collection

Note that *garbage collection* for Java components is asynchronous to program execution and can cause unexpected behavior. This behavior especially affects MTS components that perform functions such as enumerating through the collections in the catalog because the collection count will be too high (garbage collection is not synchronized). To force synchronous release of references to COM or MTS objects, you can use the release method defined in class com.ms.com.ComLib.

Example:

```
Import com.ms.com.ComLib
…
ComLib.release(someMTSObject);
```

This method releases the reference to the object when the call is executed. Release the object reference when you are sure that the reference is no longer needed. Note that if you fail to release the reference, an application error is not returned. However, an incorrect collection count results because the object reference is released asynchronously when the garbage collector eventually runs.

You can also force the release of your reference and not call that released reference again.

Example:

```
myHello = null;
   System.gc();
```

Note that forcing the release of an object reference consumes extensive system resources. It is recommended that you use the release method defined in com.ms.com.ComLib class to release references to MTS objects in a synchronous fashion.

Automating MTS Deployment

This document describes how you can use the *scriptable administration objects* to automate deployment and distribution of your MTS packages. The MTS Explorer lets you configure and deploy packages by using a graphical user interface rather than by programming code. However, you can use the scriptable administrative objects to automate administration tasks, such as program configuration and deployment. Note that the scriptable administrative objects support the same collection hierarchy as the MTS Explorer. The following figure shows the MTS Explorer collection hierarchy.

For more information about MTS Explorer functionality, see the *Administrator's Guide*.

Using the Scriptable Administration Objects

Microsoft Transaction Server contains Automation objects that you can use to program administrative and deployment procedures, including:

- Installing a Pre-Built Package
- Creating a New Package and Installing Components
- Enumerating Through Installed Packages to Update Properties
- Enumerating Through Installed Packages to Delete a Package
- Enumerating Through Installed Components to Delete a Component
- Accessing Related Collection Names
- Accessing Property Information
- Configuring a Role
- Exporting a Package
- Configuring a Client to Use Remote Components

Note that you can use the scriptable administration objects to automate any task in the MTS Explorer.

The scriptable administration objects are derived from the **IDispatch** interface, so you can use any Automation language to develop your package, such as Microsoft® Visual Basic® version 5.0, Microsoft Visual C++® version 5.0, Microsoft Visual Basic® Scripting Edition (VBScript), and Microsoft JScript™.

Each folder in the MTS Explorer hierarchy corresponds to a collection stored in the *catalog data store*. The following scriptable objects are used for administration:

- Catalog
- CatalogObject
- CatalogCollection
- PackageUtil
- ComponentUtil
- RemoteComponentUtil

The Catalog, CatalogObject, and CatalogCollection scriptable objects provide top-level functionality such as creating and modifying objects. The Catalog object enables you to connect to specific servers and access collections. Call the CatalogCollection object to enumerate, create, delete, and modify objects, as well as to access related collections. CatalogObject allows you to retrieve and set properties on an object. The Package, Component, Remote Component, and Role objects enable more specific task automation, such as installing components and exporting packages. This utility layer allows you to program very specific tasks for collection types, such as associating a role with a user or class of users.

The following diagram illustrates how the MTS scriptable administration objects interact with the MTS

Explorer catalog:

Interface	Description
ICatalog	The Catalog object enables you to connect to specific servers and access collections.
ICatalogCollection	The CatalogCollection object can be used to enumerate objects, create, delete, and modify objects, and access related collections.
ICatalogObject	The CatalogObject object provides methods to get and set properties on an object.
IPackageUtil	The IPackageUtil object enables a package to be installed and exported within the Packages collection.
IComponentUtil	The IComponentUtil object provides methods to install a component in a specific collection and to import components registered as an in-process server.
IRemoteComponentUtil	You can use the IRemoteComponentUtil object to program your application to pull remote components from a package on a remote server.
IRoleAssociationUtil	Call methods on the IRoleAssociationUtil object to associate roles with a component or component interface.

For example, you can automate creating a new package and installing components into the new package by using the scriptable objects in the utility layer (Package, Component, Remote Component, and Role objects).

The following Visual Basic sample shows how to use the scriptable administration objects to create and install components into a new package named "My Package."

1 Declare the objects that you will be using to create and install components into a new package.

```
Dim catalog As Object
Dim packages As Object
Dim newPack As Object
Dim componentsInNewPack As Object
Dim util As Object
```

2 Use the **On Error** statement to handle run-time errors if a method returns a failure HRESULT. You can test and respond to MTS trappable errors using the **On Error** statement and the **Err** object.

```
On Error GoTo failed
```

3 Call the **CreateObject** method to create an instance of the Catalog object. Retrieve the top level Packages collection from the CatalogCollection object by calling the **GetCollection** method. Then call the **Add** method to add a new package.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set newPack = packages.Add
Dim newPackID As String
```

4 Set the package name to "My Package" and save changes to the **Packages** collection.

```
newPackID = newPack.Key
newPack.Value("Name") = "My Package"
packages.savechanges
```

5 Call the **GetCollection** method to access the ComponentsInPackage collection. Then instantiate

the ComponentUtil object in order to call the **InstallComponent** method to populate the new package with components.

```
Set componentsInNewPack =
    packages.GetCollection("ComponentsInPackage",
    newPackID)
Set util = componentsInNewPack.GetUtilInterface
util.InstallComponent"d:\dllfilepath", "", ""
Exit Sub
```

6 Use the **Err** object to display an error message if the installation of the package fails.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

For a complete description of how to program these procedures and more sample code, refer to the *Administrator's Guide*.

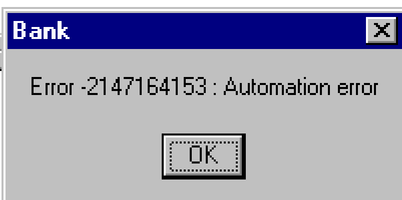
MTS Error Diagnosis

This topic describes how to determine the source of an error in your MTS application. You can diagnose the source and obtain a description of application errors by using a combination of Microsoft® Windows NT®, MTS, and other tools. If you discover that the application error is caused by MTS, you can interpret the error message using the Win32 (Win32.h) or MTS header files (mts.h), or the Microsoft Visual C++® error utility.

For more information on debugging an MTS application, see Debugging MTS Components.

Finding the Source of the Error

If your server application is failing or causing unexpected behavior, you must first determine where your error occurred. Windows NT provides a system Event Viewer that tracks application, security, and system events. Refer to the Application Log in the Event Viewer first to check the application associated with the event message. (Because you can also archive event logs, you can track an event history of the error.) Selecting an entry in your log activates an Event Detail, which provides further information about the system event. If you attempt to run the Sample Bank client without starting the Microsoft Distributed Transaction Coordinator (MS DTC), you will be returned the following Automation error.



Since this error does not indicate which application caused the failure, you can reference the Application Log in the Event viewer, which shows the error was caused by MTS.

Event Viewer - Application Log on \\DJENNE2

Log View Options Help

Date	Time	Source	Category	Event		
6/12/97	4:37:54 PM	Transaction Ser	Executive	4139		
6/12/97	4:37:54 PM	Transaction Ser	Executive	4138		
6/12/97	4:37:02 PM	Transaction Ser	Executive	4139		
6/12/97	4:37:02 PM	Transaction Ser	Executive	4138		
6/12/97	4:36:51 PM	MSDTC	SVC	4111		
6/12/97	4:28:30 PM	LicenseService	None	213		
6/12/97	4:13:28 PM	LicenseService	None	213		
6/12/97	3:58:23 PM	LicenseService	None	213		
6/12/97	3:43:18 PM	LicenseService	None	213		
6/12/97	3:28:18 PM	LicenseService	None	213		
6/12/97	3:13:17 PM	LicenseService	None	213		
6/12/97	2:58:16 PM	LicenseService	None	213		
6/12/97	2:43:15 PM	LicenseService	None	213		
6/12/97	2:27:51 PM	LicenseService	None	213		
6/12/97	2:12:52 PM	LicenseService	None	213		
6/12/97	1:57:50 PM	LicenseService	None	213		
6/12/97	1:42:48 PM	LicenseService	None	213	N/A	DJENN
6/12/97	1:27:48 PM	LicenseService	None	213	N/A	DJENN
6/12/97	1:12:47 PM	LicenseService	None	213	N/A	DJENN
6/12/97	12:57:47 PM	LicenseService	None	213	N/A	DJENN
6/12/97	12:42:42 PM	LicenseService	None	213	N/A	DJENN

Event Detail

Date: 6/12/97
Time: 4:37:02 PM
User: N/A
Computer: DJENNE2

Description:
An error occurred when starting a
to that object and other related obj

Data: Bytes Words

0000: 05 40 00 80

Close Previous

Note If you are using MTS for Windows 95, events are written to text files in the \Windows\ MTSLogs directory.

Interpreting Error Messages

The Event Viewer helps you determine the application source of the problem. You can use other tools to interpret individual error messages. Success, warning, and error values are returned using a 32-bit number known as a *result handle*, or HRESULT. HRESULTs are 32-bit values with several fields encoded in the value. A zero result indicates failure if that bit is set. A non-zero result can be a warning or informational message.

HRESULTs work differently, depending on the platform you are using. On 16-bit platforms, an HRESULT is generated from a 32-bit value known as a *status code*, or SCODE. On 32-bit platforms, an HRESULT is the same as an SCODE. Note that if you are returning HRESULTs from Java, you should throw an instance of com.ms.com.ComFailException to indicate failure. You can specify a particular HRESULT when constructing the ComFailException object. The HRESULT is used as the return value for the COM method. To indicate successful completion, you do not need to do anything; just return normally. To return S_FALSE, indicating a successful completion but a return value of Boolean False, throw an instance of com.ms.com.ComSuccessException. In Visual Basic, you use the **Err.Raise** function to set and the **On Error**… / **Err.Number** to retrieve HRESULTs.

For a list of the values of common system-defined HRESULTs, see ComFailException. For a

complete list of system-defined HRESULT values, see the header file Winerror.h included with the Platform SDK.

MTS never changes the value of an HRESULT error code, such as E_UNEXPECTED or E_FAIL, returned by an MTS object method. When an MTS object returns an HRESULT status code (such as S_OK or S_FALSE), MTS may convert the status code into an MTS error code before it returns to the caller. This occurs, for example, when the application returns S_OK after calling the **SetComplete** function; if the object is the root of an automatic transaction that fails to commit, the HRESULT is converted to CONTEXT_E_ABORTED. When MTS converts a status code to an *error code*, all the method's output parameters are cleared. Returned references are released and the values of the returned object pointers are set to NULL.

The Mtx.h header file contains the MTS specific error codes. Winerror.h contains the error code definitions for the Win32 API. For an overview of error codes, see "Error Handling" in the COM portion of the Microsoft Platform SDK

You can also use the ERRLOOK utility in Microsoft Visual Studio™ 97 to retrieve a system error message or module error message based on the value entered. ERRLOOK retrieves the error message text automatically if you drag-and-drop a hexadecimal or decimal value from the Visual Studio debugger or other Automation-enabled application. You can also enter a value either by typing it in or pasting it from the IDE clipboard and clicking the **Look Up** option.

Contacting MTS Support

If you run into a problem that you cannot solve, you can contact Microsoft support with the following information:

- Topography of the application where the error occurred, such as a description of packages, components, and interfaces
- Application event log on all computers
- Reproduction of the error, if possible

Troubleshooting

If you are having trouble diagnosing your problem, refer to the list of troubleshooting tips below:

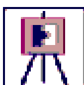
- Make sure that the Distributed Transaction Coordinator (DTC) is running on all servers.
- Check network communication by first testing on a local computer to verify that the application works. If you are running TCP/IP on your network, you can then use the Windows NT Ping.exe utility to verify that the machines are on the network.
- Make sure that SQL and DTC are either located on the same computer or that the DTC Client Configuration program specifies that the DTC is on another computer. If not, **SQLConnect** will return an error internally when called from a transactional component.
- Set the MTS transaction timeout to a higher number than the default 60 seconds, otherwise MTS aborts the transaction after this time has lapsed. All subsequent calls to the component return immediately with CONTEXT_E_ABORTED.
- Make sure that your ODBC drivers are thread-safe and do not have thread affinity.
- If you have difficulty getting an application to work over several servers, reboot the client and then verify that your Windows NT domain controller is configured properly.
- Turning off resource pooling may reveal that a resource dispenser used by your application is the source of the problem. You can turn off resource pooling by setting the following registry key:

  HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Transaction Server\Local Computer\My Computer:Resource Pooling=N

See the DCOM documentation and DCOM-related Knowledge Base articles if you are experiencing application errors that you suspect are caused by DCOM.

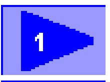
Creating a Simple ActiveX Component

This section gets you started quickly with a simple ActiveX™ component (Account). You then install, run, and monitor Account with the Microsoft Transaction Server (MTS) Explorer and a sample client (Bank).



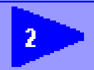
Scenario: Creating and Using a Simple ActiveX Component

First, create your component (Account) and run it in the MTS run-time environment by using the Bank client. Then, add a database connection to Account and run it again.



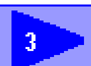
Creating the Account Component

Create a new ActiveX component, Account.



Creating the Bank Package

Use the MTS Explorer to create a new package for your component.

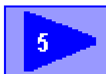


Installing the Account Component in the Bank Package

Use the MTS Explorer to install your component in a package.

Running and Monitoring the Account Component

Use the Bank client to run your component, and use the MTS Explorer to monitor it.



Modifying the Account Component: Add a Database Connection

Modify your component so that it connects to a database. The connection will be pooled by the ODBC resource dispenser. Then use the Bank client to re-run the modified component.

Application Design Notes: Sharing Resources

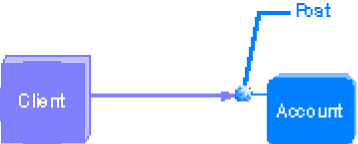
Learn how to efficiently share resources, such as database connections, through the MTS Resource Dispenser Manager and resource dispensers.

See Also

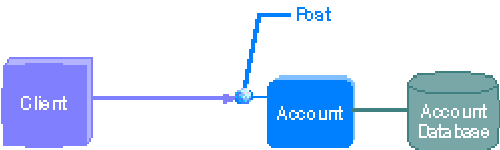
Programming Concepts, Transaction Server Components, Building Scalable Components

Scenario: Creating and Using a Simple ActiveX Component

This scenario has two implementation stages. The initial stage consists of building a simple component: the Account component. You can implement Account by creating a project (Account.vbp) with a class module (Account.cls). The Account class module exposes one method, **Post**, that passes back a string indicating that it was called successfully. The following illustration depicts the first stage of this scenario.



The second stage of this scenario adds a database connection to get the appropriate account information from a database and update it. This demonstrates using a resource dispenser in this case, the ODBC resource dispenser, which enables efficient connection pooling. The following illustration depicts the second stage of this scenario.



The rest of this section provides step-by-step instructions for creating, installing, and running the Account component in this scenario. You can find the Microsoft Visual Basic projects for each of these steps in the Step1 through Step8 folders in the \Samples\Account.VB folder of your Microsoft Transaction Server installation.

See Also

Programming Concepts, Application Design Notes: Sharing Resources, Transaction Server Components, Building Scalable Components, Creating the Account Component, Run and Monitor the Account Component, Modifying the Account Component, Application Design Notes: Resource Usage, Creating a Simple ActiveX Component

Creating the Account Component

The first step toward building a simple application that you can use with Microsoft Transaction Server (MTS) is to create a simple ActiveX™ component DLL (VBAcct.dll); the Account component provides one method, Post.

The information presented here assumes a basic understanding of how to use Microsoft Visual Basic to create ActiveX components.

Note You cannot install executable files (.exe) in MTS. If you have a component built as an executable file, you must rebuild it as a dynamic-link library (DLL).

To create the Account component

1. Start Microsoft® Visual Basic™ and open the \Mts\Samples\Account.VB\Step1\Account.vbp project.

   Click here to see the code for the Account component

2. Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step1\VBAcct.dll.

See Also

Programming Concepts, Transaction Server Components, Building Scalable Components, Transaction Server Component Requirements, Run and Monitor the Account Component, Application Design Notes: Sharing Resources, Modifying the Account Component, Application Design Notes: Resource Usage, Creating a Simple ActiveX Component

Creating the Bank Package

To run your component in the MTS run-time environment, you need to create a package. For this scenario, you will create a package with a single component.

A package is a collection of components that you can deploy and manage as a unit. By grouping components into packages, you define the security and process boundaries for components running on a computer. The criteria for deciding how to group components into packages require achieving the optimum balance between performance, fault isolation, and load balancing.

You will create a package called Bank that will contain the Account component.

To create the Bank package

1. On the **Start** menu, point to **Programs**, point to **Microsoft Transaction Server**, and then click **Transaction Server Explorer**.
2. Create a new package named **Bank**. In the **Set Package Identity** dialog box, select **Interactive user**.

How?

You can use the **General**, **Security**, **Advanced**, **Identity, and Activation** tabs to configure a package. For this scenario, you will use the default settings for the package you just created.

See Also

What Does Creating a Package Mean?, Package Properties , Programming Concepts, Installing the Account Component in the Bank Package, Creating a Simple ActiveX Component

Installing the Account Component

To run your components in the Microsoft Transaction Server run-time environment, you first need to install them in a package. This means you need to install the Account component in the Bank package.

To install the Account component

- Install the Account component into the Bank package you created in Creating the Bank Package. Use the Account component that you built in \Mts\Samples\Step1\Account.VB\VBAcct.dll.

How?

You can use the **General**, **Transaction**, and **Security** tabs to configure a component. For this scenario, you will use the default settings for the component you just installed.

See Also

Adding A Component to a Package, Component Properties , Creating the Bank Package, Creating a Simple ActiveX Component

Running and Monitoring the Account Component

Now that you have created the Bank package and installed the Account component in the Microsoft Transaction Server Explorer, you can run the Account component with the Bank client and monitor the component status in the Explorer.

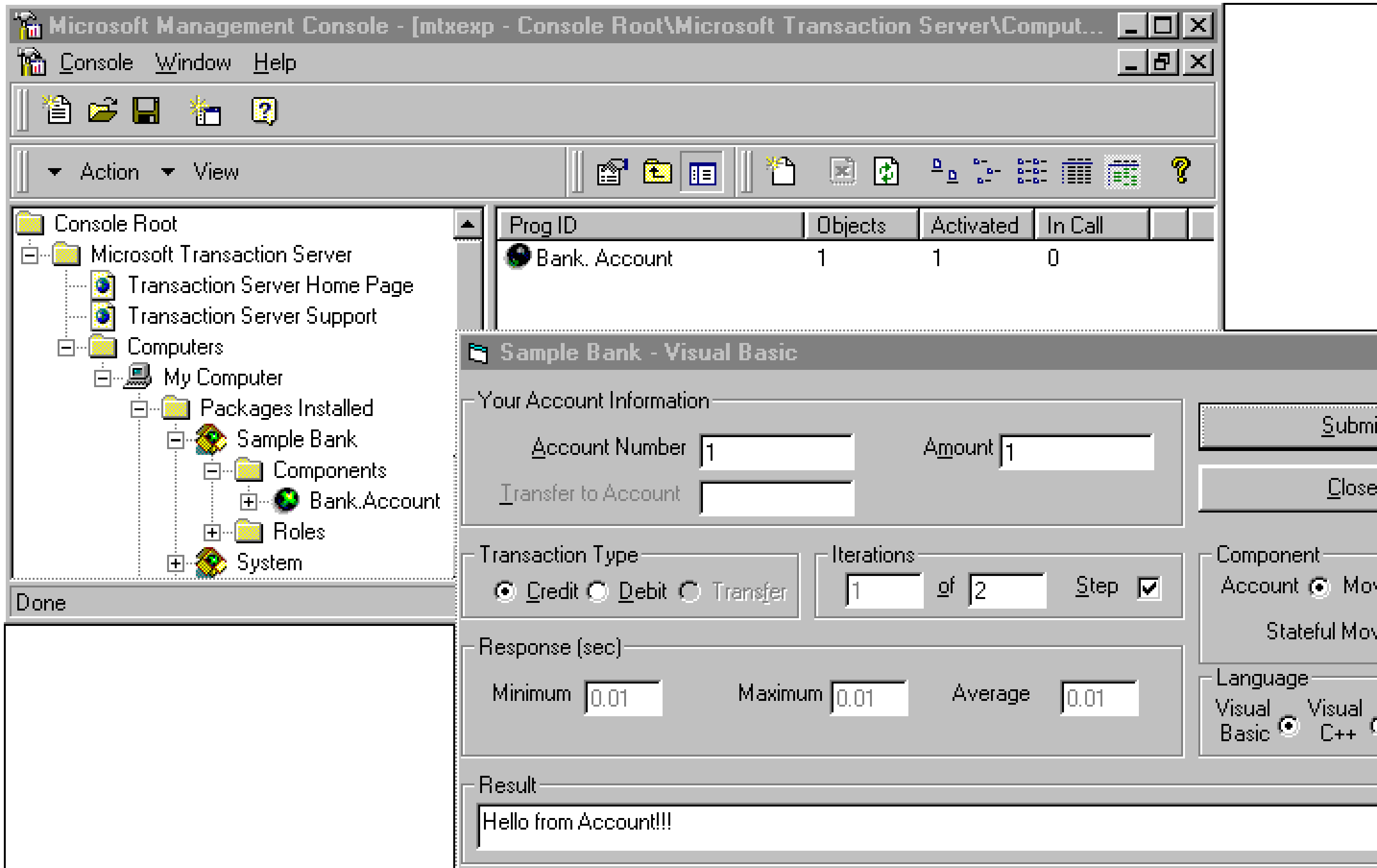
Running your component in MTS brings immediate benefits to your application, even if it doesn't implement any of the MTS APIs. Such benefits include:

- Simplified management of your components though an easy-to-use graphical tool, the MTS Explorer.
- Location transparency—the ability to run your components in-process, locally, or remotely.
- Thread management and component tracking.
- Database connection pooling through the Resource Dispenser Manager and the ODBC resource dispenser (automatically provided if you call the ODBC API).

To run and monitor your component

1. In the left pane of the MTS Explorer, click the Components folder where you installed the Bank.Account component.
2. On the **View** menu, click **Status view** to display usage information for the Bank.Account component.
3. On the **Start** menu, point to **Programs**, point to **Microsoft Transaction Server**, and then click **Bank Client**.

   Arrange the windows so that you can see the Bank Client window and the MTS Explorer window simultaneously. The form will default to credit $1 to account number 1.



4 In the Bank client, click the **Account** component.

5 Click **Submit**.

You should see the response **Hello from Account**.

6 In the Bank client, change the iterations from **1 of 0** to **1 of 100** and click **Submit**.

In the right pane of the MTS Explorer, you should see the values under the **Objects** and **Activated** columns change to 1 and back to 0.

See Also

Status View, Creating the Bank Package, Installing the Account Component in the Bank Package, Microsoft Transaction Server APIs, Maintaining MTS Packages, Creating a Simple ActiveX Component

Modifying the Account Component to Use the ODBC Resource Dispenser

In this section, you enhance the Account component by adding a database connection. You revise the Post method to access a database using ActiveX Data Objects (ADO) to obtain the appropriate account information. This demonstrates using a resource dispenser in this case, the ODBC resource dispenser, which enables connections to be pooled for efficiency. You will also add a new class module, CreateTable, to the Account project.

To modify the Account component

1 Open the \Mts\Samples\Account.VB\Step2\Account.vbp project.

Click here to see the modified Account component

2 Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step2\VBAcct.dll.

By adding a new class module, you have added a new COM component to this DLL. Therefore, you will need to delete the Account component in the Microsoft Transaction Server Explorer and then install the Account and the MoveMoney components.

To reinstall your components

1 Remove the Account component.

How?

2 Add the new components.

How?

Use the DLL you created in \Mts\Samples\Account.VB\Step2\VBAcct.dll.

You can now run the new component by using the Bank client. You should see a response **Credit, balance is $ 1. (VB)**.

See Also

Creating the Account Component, Application Design Notes: Resource Usage, Programming Concepts, Creating a Simple ActiveX Component

Application Design Notes: Sharing Resources

Each time the **Account** object's **Post** method is called, it obtains, uses, and then releases its database connection. A database connection is a valuable resource. The most efficient model for resource usage in scalable applications is to use them sparingly, acquire them only when you really need them, and return them as soon as possible.

Historically, acquiring resources has been an expensive operation in terms of system performance. Many programs have adopted a strategy of acquiring resources and holding onto them until program termination. While this strategy is effective for single-user systems, building scalable server applications requires sharing these resources.

Microsoft Transaction Server provides an architecture for resource sharing through its Resource Dispenser Manager and resource dispensers. The Resource Dispenser Manager works with specific resource dispensers to automatically pool and recycle resources. The ODBC version 3.0 Driver Manager is a Microsoft Transaction Server resource dispenser, also referred to as the ODBC resource dispenser.

Although the Account component hasn't implemented any of the MTS-specific APIs, when you run it, MTS uses the ODBC resource dispenser. This happens automatically when the Post method uses ActiveX Data Objects (ADO) to access the database, because ADO in turn uses ODBC. Whenever any component running in the MTS run-time environment uses ODBC directly or indirectly, the component automatically uses the ODBC resource dispenser.

When the **Account** object releases the database connection, the connection is returned to a pool. When the **Post** method is called again, it requests the same database connection. Instead of creating a new connection, the ODBC resource dispenser recycles the pooled connection, which saves time and server resources.

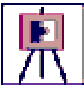
The topic Building Scalable Components, shows you how to use just-in-time activation to use server resources even more efficiently, resulting in more scalable applications and improved performance.

See Also

Application Design Notes: Resource Usage, Building Scalable Components, Modifying the Account Component, Creating the Account Component, Creating a Simple ActiveX Component

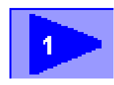
Building Scalable Components

In this section, you'll learn how you can use just-in-time activation to use server resources efficiently, resulting in more scalable applications and improved performance. You'll also see how a simple change to the Account component allows it to scale efficiently and support a large number of clients, without requiring you to make any changes to the client.



Scenario: Adding Just-In-Time Activation to the Account Component

Add code to your Account component to take advantage of just-in-time activation, which releases the component's resources when Account has completed its work.



Adding Code to Call GetObjectContext, SetComplete, and SetAbort

Add code to call **GetObjectContext**, **SetComplete**, and **SetAbort**.

Application Design Notes: Just-In-Time Activation

Learn how to reuse resources efficiently so you can build scalable applications.

See Also

Context Objects, Deactivating Objects, Creating a Simple ActiveX Component, **GetObjectContext** method, **SetAbort** method, **SetComplete** method

Scenario: Adding Just-In-Time Activation to the Account Component

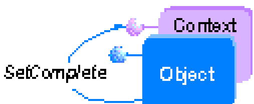
In this scenario, the Bank client creates an Account object from the Account component and calls its Post method, just as in Creating a Simple ActiveX Component. This time, Account obtains a reference to its context object. When it successfully completes its work on behalf of the client, it uses its context object to call **SetComplete**. If Account encounters an error and is unable to complete its work successfully, it uses its context object to call **SetAbort**. When Account calls either **SetComplete** or **SetAbort**, it indicates that it's finished with its work and that it doesn't need to maintain any private state for its client. This allows the MTS run-time environment to reclaim and reuse the Account object's resources.

See Also

Context Objects, Deactivating Objects, Application Design Notes: Just-In-Time Activation, **GetObjectContext** method, **SetAbort** method

Adding Code to Call GetObjectContext, SetComplete, and SetAbort

Every Transaction Server object has a context object associated with it. The context object is automatically created at the same time the object itself is created. You can use an object's context to declare when the object's work is complete, as shown in the following illustration.



Calling either of these methods notifies the MTS run-time environment that it can safely deactivate the object, making its resources available for reuse.

To implement the scenario for this chapter, you will modify the Post method to use the Account object's context object. Then you will use **SetComplete** and **SetAbort** to enable just-in-time activation.

First, you call **GetObjectContext** to get a reference to the context object. When an object has completed its work successfully, it should call **SetComplete**:

```
GetObjectContext.SetComplete
```

SetComplete notifies the MTS run-time environment that the Account object should be deactivated as soon as it returns control to the Bank client.

If the object encountered an error, it should call **SetAbort**. **SetAbort** also notifies the MTS run-time environment that the Account object should be deactivated as soon as it returns control to the Bank client.

```
GetObjectContext.SetAbort
```

To obtain a reference to an object's context

1. Open the \Mts\Samples\Account.VB\Step3\Account.vbp project.

Click here to see the Post method

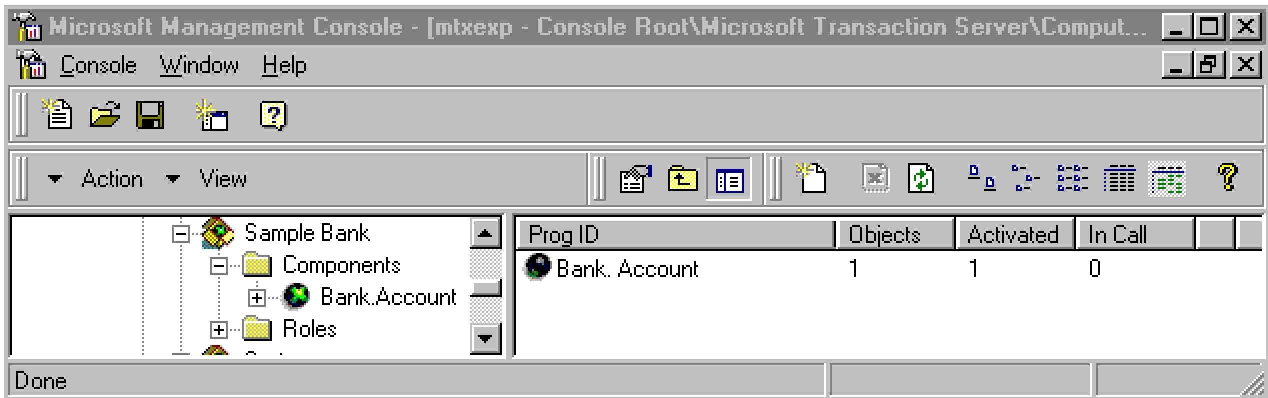
2. Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step3\VBAcct.dll.

Before you can run your new component again in MTS, the registry needs to be updated with the new component information. To do this, refresh the MTS Explorer window.

If you install the Development version of Microsoft Transaction Server, you will get a Visual Basic – compatible add-in that automates this process for you (select the VB Addin box during Setup). The next time you run Visual Basic, the add-in is automatically installed in Visual Basic. The add-in automatically refreshes all of your MTS component DLLs whenever you recompile your project.

You can also turn this feature on and off on a per-project basis by using the toggle command on the Visual Basic **Add-Ins** menu. To turn it on, on the Visual Basic **Add-Ins** menu, point to **MS Transaction Server**, and click **AutoRefresh after compile of active project**. This puts a check mark next to the command, indicating that the feature is activated. If you want to refresh all of your MTS components at any given time, on the Visual Basic **Add-Ins** menu, point to **MS Transaction Server**, and then click **Refresh all components now**.

Now you'll run the Account component again from the Bank client, and monitor its execution in the MTS Explorer's Status window. Follow the same steps as in "Running and Monitoring the Account Component."



When the Bank client creates the Account object, the number 1 will appear under **Objects** and **Activated**. This indicates that one object is executing in the MTS run-time environment, and that it is currently activated. When the client calls the Post method, the number 1 appears, briefly, under **In Call**. This indicates that one object is currently executing a method call. When the Post method returns control to the client, the number under **Objects** is still 1, but the numbers under **Activated** and **In Call** return to 0. This is because after calling **SetComplete**, the object is deactivated as soon as it returns from the current method call.

Note Because the Post method executes so quickly, you may not actually see this sequence appear.

See Also

Context Objects, Deactivating Objects, Creating a Simple ActiveX Component, **GetObjectContext** method, **SetAbort** method, **SetComplete** method

Application Design Notes: Just-In-Time Activation

When you design a traditional application, you have two options:

- A client can create, use, and release an object. The next time it needs the object, it creates it again.

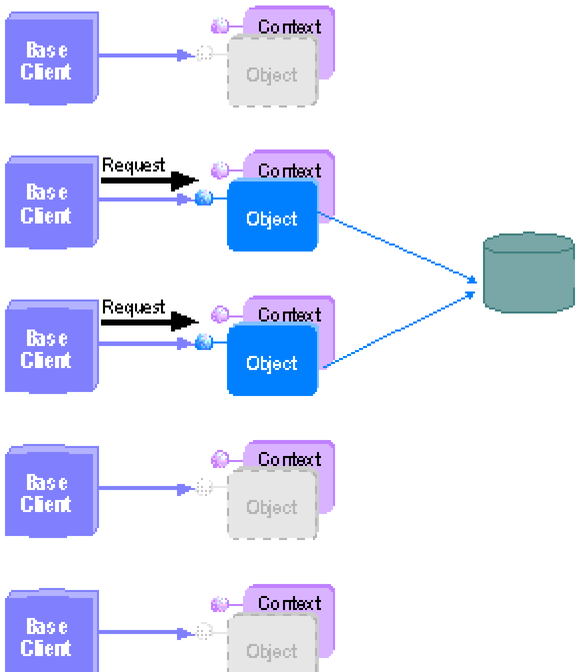
  The advantage to this technique is that it conserves server resources. The disadvantage is that, as your application scales up, your performance slows down. If the object is on a remote computer, each time an object is created, there must be a network round-trip, which negatively affects performance.
- A client can create an object and hold onto it until the client no longer needs it.

  The advantage of this approach is that it's faster. The problem with it is that, in a large-scale application, it quickly becomes expensive in terms of server resources.

While either of these approaches might be fine for a small-scale application, as your application scales up, they're both inefficient. Just-in-time activation provides the best of both approaches, while avoiding the disadvantages of each.

In Creating a Simple ActiveX Component, the Bank client controlled the Account object's life cycle. Clients held onto server resources even when the clients were idle. As you added more clients, you saw a proportional increase in the number of allocated objects and database connections. A quick look at the Account component shows that each call to the Post method is independent of any previous calls. An Account object doesn't need to maintain any private state to correctly process new requests from its client. It also doesn't need to maintain its database connection between calls. The only problem is that, in this scenario, the MTS run-time environment can't reclaim the object's resources until the client explicitly releases the object. If you have to depend on your clients to manage your object's resources, you can't build a scalable component.

By adding just a few lines of code, you were able to implement just-in-time activation in the Account component. When an Account object calls **SetComplete**, it notifies the MTS run-time environment that it should be deactivated as soon as it returns control to the client. This allows the MTS run-time environment to release the object's resources, including any database connection it holds prior to the object's release. The Bank client continues to hold a reference to the deactivated Account object.



When a client calls a method on a deactivated object, the client's reference is automatically bound to a new object. Thus, the client has the illusion of a continuous reference to a single object, without

tying up server resources unnecessarily.

Although the call to **SetAbort** has a similar effect, it isn't apparent in this scenario why it is used when errors occur. The next chapter, Building Transactional Components, shows you how transactions can make your applications more robust in the event of an error.

See Also

Context Objects, Deactivating Objects, Creating a Simple ActiveX Component, **GetObjectContext** method, **SetAbort** method, **SetComplete** method

Building Transactional Components

This section introduces transactional components and the benefits of running components within the same transaction.

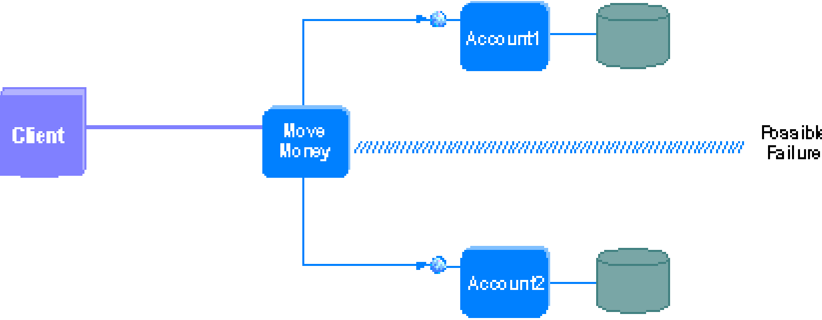
- **Scenario: Composing Work from Multiple Components Under the Same Transaction**
  Add new functionality to transfer money between accounts by adding a new component, MoveMoney, which uses the existing Account component.
- **Creating the MoveMoney Component**
  Use the **CreateInstance** method to run the MoveMoney and Account components within the same transaction.
- **Monitoring Transactions**
  Use the Bank client to run your components, and use the Microsoft Transaction Server Explorer to monitor transactions.
- **Application Design Notes: Using Context and Transactions**
  Using transactional components provides atomicity and simplified error recovery.

See Also

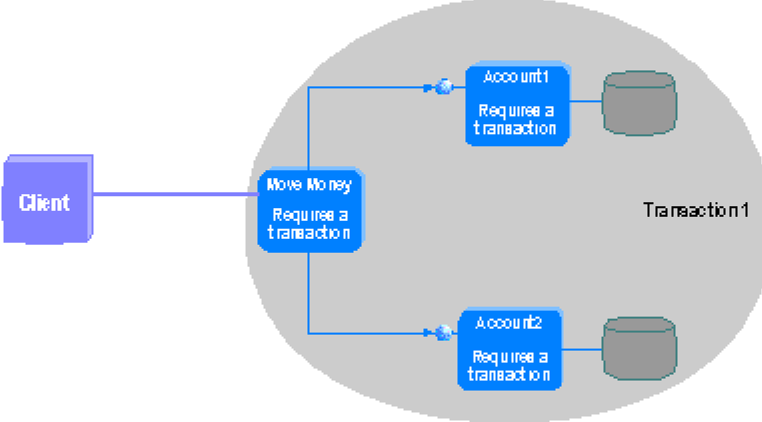
Transactions, Transaction Attributes, **ObjectContext** object, **CreateInstance** method

Scenario: Composing Work from Multiple Components Under the Same Transaction

For this scenario, you will add new functionality that allows you to transfer money between two accounts. To implement this, you add a new component, MoveMoney. MoveMoney creates Account and then calls it once for a credit or debit, or twice for a transfer, as shown here.



Because either MoveMoney or Account could fail at any point, all database updates need to be in the same transaction to ensure that the database remains consistent. To do this, you configure the MoveMoney and Account components in the Microsoft Transaction Server Explorer to require a transaction. Transaction Server ensures that all their objects' work is automatically done in the same transaction.



See Also

Transactions, Transaction Attributes

Creating the MoveMoney Component

To implement this scenario, you will add a new class module, MoveMoney, to the Account project. MoveMoney has a single method, Perform, which creates an Account object to perform the credit, debit, or transfer.

To create the MoveMoney component

1. Open the \Mts\Samples\Account.VB\Step4\Account.vbp project.

Click here to see the Perform method

2. Build the component as a dynamic-link library (DLL) and save it as \Mts\Samples\Account.VB\Step4\VBAcct.dll.

By adding a new class module, you have added a new COM component to this DLL. Therefore, you will need to delete the Account component in the Microsoft Transaction Server Explorer and then install the Account and the MoveMoney components.

To reinstall your components

1. Remove the Account and CreateTable components.

How?

2. Add the new components.

How?

Use the DLL you created in the previous procedure. You can find it in \Mts\Samples\Account.VB\Step4\VBAcct.dll.

MTS enlists a component in a transaction as specified by the component's transaction attribute. For this scenario, Account and MoveMoney run within the same transaction.

To set the transaction attributes for your components

1. For the Account and MoveMoney components, set the transaction attribute to **Requires a transaction**.
2. For the CreateTable component, set the transaction attribute to **Requires a new transaction**.

How?

The MoveMoney object uses **CreateInstance** to create the Account object. **CreateInstance** is a method on the context object. By using **CreateInstance**, the Account object created by MoveMoney shares context with MoveMoney.

```
Dim objAccount As Bank.Account
Set objAccount = _
    GetObjectContext.CreateInstance("Bank.Account")
```

Transactions are associated with an object's context. Because both MoveMoney and Account have a transaction attribute of **Requires a transaction**, the Account object will be enlisted within the same transaction as MoveMoney.

In Building Scalable Components, you learned how to use **SetComplete** to indicate that an object has finished its work and can be deactivated. For transactional components, calling **SetComplete** indicates that a transaction can be committed.

```
GetObjectContext.SetComplete
```

When the Perform method returns, the transaction attempts to commit. There is no guarantee that it will commit, however. If an error occurs, Perform instead calls **SetAbort**.

```
GetObjectContext.SetAbort
```

SetAbort also indicates that an object has finished its work, but that it isn't in a consistent state. When the Perform method returns after calling **SetAbort**, the attempt to commit the transaction won't succeed.

See Also

Transactions, Transaction Attributes, Context Objects, Creating MTS Objects, **ObjectContext** object, **CreateInstance** method, **GetObjectContext** method, **SetAbort** method, **SetComplete** method

Monitoring Transaction Statistics

You can use the Microsoft Transaction Server Explorer to monitor commit and abort statistics for transactions.

You can experiment with the MoveMoney and Account components to see how transactions are committed and aborted as you provide user input with the Bank client.

To monitor transactions

1. On the **Window** menu of the Transaction Server Explorer, click **New Window**.
2. In the left pane, click **Transaction Statistics**.
3. On the **Action** menu, click **Scope Pane** to hide the left pane of the Explorer.
4. Make sure that the Microsoft Distributed Transaction Coordinator (MS DTC) is running on your SQL Server computer. You can start MS DTC from the Transaction Server Explorer or from SQL Server.
5. Also make sure that you have the ODBC data source set up, and that SQL Server is running. Click to get information on how to do this.
6. Start the Bank client.
   Rearrange the windows so that you see the two Microsoft Transaction Server Explorer windows and the Bank client window.

To monitor a commit, click **Submit** in the Bank client. The Transaction Statistics window first indicates that one transaction is active, and indicates that one transaction was committed.

To monitor an abort, click **Debit** in the Bank client, and enter an amount for the transaction that is greater than the balance on your account. Click **Submit**. The Transaction Statistics window first indicates that one transaction is active, and then indicates that one transaction was aborted.

Try experimenting with **Transfer**. Verify that both objects are running within the same transaction by checking the balance of two accounts and performing a transfer that would overdraw from an account. Notice that both the credit and the debit are aborted.

See Also

Monitoring Transactions in MTS

Application Design Notes: Using Context and Transactions

Context simplifies defining transactions. A transaction is automatically started when a component is declared as transactional. Components don't need to add additional code to indicate the start and end of a transaction. Using context allows you to define the scope of a transaction.

Besides simplifying building components, automatic transaction enlistment also allows for reuse of existing components. Changing the transaction attribute is the only change to the Account component from the previous section, Building Scalable Components.

Creating the Account object from MoveMoney establishes MoveMoney as the root of the transaction. The root transaction attempts to commit after it has completed its work. If an Account object calls **SetAbort** to indicate that it cannot successfully commit its work, then when the root transaction attempts to commit, the entire transaction will fail.

In the case of a money transfer, this provides atomicity. If a credit succeeds, but insufficient funds prevent the debit from succeeding, then the credit will be rolled back from the database automatically. Thus, **SetAbort** provides simplified error recovery.

Context simplifies the development of the component. Each object independently acquires its own resources, performs its work, and indicates its own internal state by using **SetComplete** or **SetAbort** before returning.

See Also

Transactions, Transaction Attributes, Context Objects, **CreateInstance** method, **ObjectContext** object, **SetAbort** method, **SetComplete** method

Sharing State

This chapter shows you how to use the Shared Property Manager to share state among multiple Microsoft Transaction Server objects running in the same process.

- **Scenario: A Receipt Number Generator That Uses the Shared Property Manager**

  Create a component that uses the Shared Property Manager to generate a unique receipt number for each bank transaction.

- **Creating the Receipt Component**

  Create a **SharedProperty** object to get a new receipt number, with appropriate isolation and release modes for this scenario.

Application Design Notes: Sharing State by Using the Shared Property Manager

Learn some of the advantages of using the Shared Property Manager to manage shared state within a process.

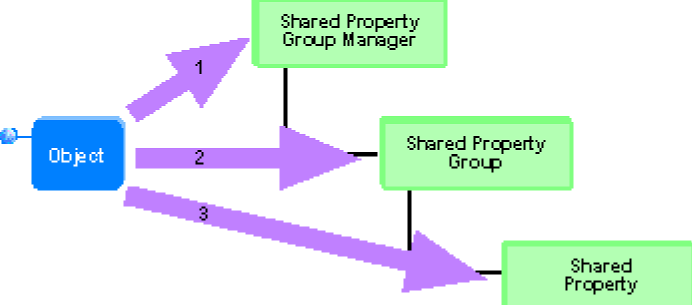
See Also

Resource Dispensers, Stateful Components, **SharedPropertyGroupManager** object

Scenario: A Receipt Number Generator That Uses the Shared Property Manager

This chapter introduces the Receipt component, which dispenses unique receipt numbers for fund transfers. When a bank transaction takes place, the MoveMoney object creates a Receipt object. The Receipt component contains a single method, GetNextReceipt.

GetNextReceipt uses the Shared Property Manager to get a unique receipt number. The Shared Property Manager has an object hierarchy as shown in the following figure:



Within a server process, there is only one instance of the **SharedPropertyGroupManager** object. The value of the receipt number is maintained by a **SharedProperty** object, which provides locking mechanisms to ensure that no two calls to GetNextReceipt retrieve the same value.

See Also

Resource Dispensers, Creating the Receipt Component, Application Design Notes: Sharing State by Using the Shared Property Manager, **SharedPropertyGroupManager** object, **SharedPropertyGroup** object, **SharedProperty** object

Creating the Receipt Component

The Receipt component contains a single method, GetNextReceipt. The Receipt object itself doesn't maintain the value of the receipt number between calls. The Shared Property Manager maintains these values. The Receipt object calls a **SharedProperty** object to get a new receipt number.

You will also add code to the MoveMoney component to call the Receipt component.

To create the Receipt Component

1 Start Microsoft Visual Basic and open the \Mts\Samples\Account.VB\Step5\Account.vbp project.

Click here to see the code for the Receipt component

Click here to see the code for the MoveMoney component

2 Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step5\VBAcct.dll.

By adding a new class module, you add a new COM component to this DLL. Therefore, you need to delete the existing components in the Microsoft Transaction Server Explorer and then install the new components.

To reinstall your components

1 Remove the Account, MoveMoney, and CreateTable components from the Transaction Server Explorer.

How?

2 Add the new components. Use the DLL you created in \Mts\Samples\Account.VB\Step5\VBAcct.dll.

How?

To set the transaction attributes for your components

1 For the Account and MoveMoney components, set the transaction attribute to **Requires a transaction**.

How?

2 For the CreateTable component, set the transaction attribute to **Requires a new transaction**.

3 For the Receipt component, set the transaction attribute to **Does not support transactions**. This is the default value.

Note that the Receipt component is not transactional because the receipts are maintained as properties in memory and aren't durable.

When you run the Bank Client, select the MoveMoney button under **Component**. You should see the response **Credit, balance is $ 1. (VB); Receipt No: #####**.

The various object creation methods for Shared Property Manager objects are designed for simplified coding. If the object doesn't exist, it will be created. If it already exists, the object is returned. GetNextReceipt makes the following method call to access the shared property group manager:

```
Set spmMgr = CreateObject _
    ("MTxSpm.SharedPropertyGroupManager.1")
```

This code works every time it is called. There is no need to check if the shared property group manager has already been created. Such behavior also ensures that only one instance of the **SharedPropertyGroupManager** object exists per server process.

For the **SharedPropertyGroup** and **SharedProperty** objects, a flag is returned to indicate whether the property group or property already exists. The following code shows how this flag is used to determine if the property needs to be initialized:

```
Set spmPropNextReceipt = _
    spmGroup.CreateProperty("Next", bResult)

' Set the initial value of the SharedProperty to
' 0 if the SharedProperty didn't already exist.
If bResult = False Then
```

```
    spmPropNextReceipt.Value = 0
End If
```

Access to shared properties is controlled through the **CreatePropertyGroup** method:

```
Set spmGroup = _
    spmMgr.CreatePropertyGroup("Receipt", _
    LockMethod, Process, bResult)
```

CreatePropertyGroup has two parameters, isolation mode and release mode. The isolation mode for the Receipt property group is set to **LockMethod**, which ensures that two instances of the Receipt object can't read or write to the same property during a call to GetNextReceipt. The release mode for the Receipt property group is set to **Process**, which maintains the property group until the server process is terminated.

See Also

Application Design Notes: Sharing State by Using the Shared Property Manager, **SharedPropertyGroupManager** object, **CreateProperty** methodasmthCreatePropertyvb, **CreatePropertyGroup** methodasmthCreatePropertyGroupvb

Application Design Notes: Sharing State by Using the Shared Property Manager

Using the Shared Property Manager makes sharing state in a multiuser environment as easy as it is in a single-user environment. Without the use of the Shared Property Manager, the application would require much more code that has nothing to do with the business problem at hand.

One alternate way to create the same functionality would be to maintain the receipt number as a member variable of the Receipt component. However, this complicates coding the Receipt component immensely. The Receipt component would have to remain persistent in memory during the life of the server process. This would require the following additional code:

- Referencing all instances of MoveMoney to the Receipt object.
- Maintaining a locking mechanism to prevent concurrent access to the Receipt object.

Even after adding this code, the application wouldn't be extensible for additional shared properties. The Shared Property Manager is another example of how Microsoft Transaction Server provides the infrastructure for server applications so that you can concentrate on coding business logic.

Location Transparency and the Shared Property Manager

For objects to share state, they all must be running in the same server process with the Shared Property Manager.

To maintain location transparency, it's a good idea to limit the use of a shared property group to objects created by the same component, or to objects created by components implemented within the same DLL. When components provided by different DLLs use the same shared property group, you run the risk of an administrator moving one or more of those components into a different package. Because components in different packages generally run in different processes, objects created by those components would no longer be able to share properties.

See Also

Resource Dispensers, Stateful Components, **SharedPropertyGroupManager** object

Stateful Components

This section discusses stateful components and outlines some of the issues associated with writing stateful application components.

Scenario: Holding State in the MoveMoney Component

Consider the design alternative of holding state within objects.

Adding a New Method to the MoveMoney Component

Add the StatefulPerform method, which uses MoveMoney to maintain account number values.

Application Design Notes: The Trade-offs of Using Stateful Objects

Learn how holding state in objects affects the application behavior within the Microsoft Transaction Server run-time environment.

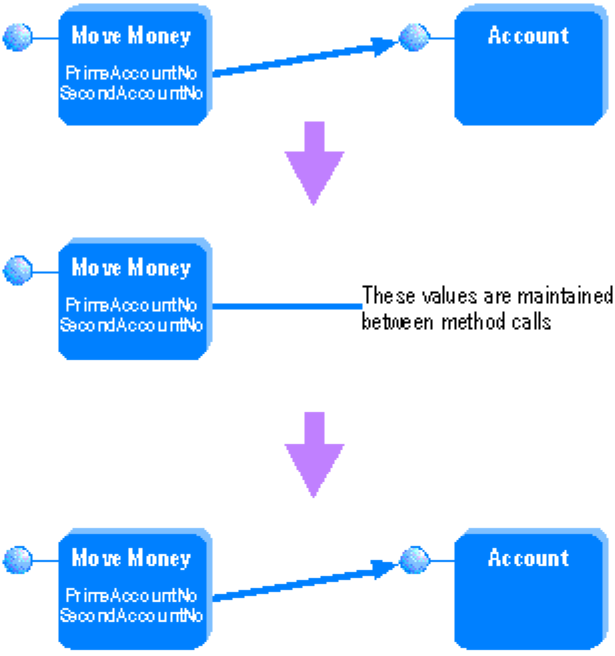
See Also

Transactions

Scenario: Holding State in the MoveMoney Component

Building stateful objects is a useful approach in application design. However, such design can have performance trade-offs. This section demonstrates how holding state in objects affects the application behavior within the Microsoft Transaction Server run-time environment.

You will modify the MoveMoney component to be stateful by adding the StatefulPerform function to MoveMoney. StatefulPerform is called when you click **Stateful MoveMoney** on the Sample Bank client. This new function causes MoveMoney to retain data in member variables between method calls.



See Also

Transactions, Deactivating Objects, Context Objects, Stateful Components

Adding a New Method to the MoveMoney Component

To implement the scenario for this section, you will add a method similar to Perform, named StatefulPerform, which uses class member variables to set account numbers. Thus, MoveMoney becomes a stateful object when StatefulPerform is called.

To add a new function to the MoveMoney component

1 Open the \Mts\Samples\Account.VB\Step6\Account.vbp project.

Click here to see the StatefulPerform method

2 Build the component as a dynamic-link library (DLL) and save it as \Mts\Samples\Account.VB\Step6\VBAcct.dll.

The code for StatefulPerform calls the Perform method. The methods differ in how the account numbers are set. Class member variables for each account must be set before calling StatefulPerform, whereas Perform passes the account numbers by value through function parameters.

When you click the **MoveMoney** option in the Sample Bank client, it calls the following code to initialize the function:

```
StatefulPerform = Perform(PrimeAccount, SecondAccount, lngAmount,
lngTranType)
```

When you click the **Stateful MoveMoney** option, the Sample Bank client calls the following code to initialize the function:

```
obj.PrimeAccount = PrimeAcct
obj.SecondAccount = lSecondAcct
Res = obj.StatefulPerform(CLng(Amount), TranType)
```

The PrimeAccount and SecondAccount properties are actually separate class member variables on the MoveMoney object. Note that the PrimeAccount and SecondAccount properties aren't accessed through the Shared Property Manager properties; the MoveMoney object controls getting and setting the account number values, thus making the MoveMoney object stateful.

Run the Bank client with the **MoveMoney** option. Then run it again with the **Stateful MoveMoney** option. You should notice that the stateless version is slightly faster. Try running multiple Bank clients with concurrent transactions. You should notice that the stateless version performs significantly better. The next section explains why.

See Also

Transactions, Deactivating Objects, Context Objects, Stateful Components

Application Design Notes: The Trade-offs of Using Stateful Objects

This section explains the trade-offs of using stateful objects in your applications.

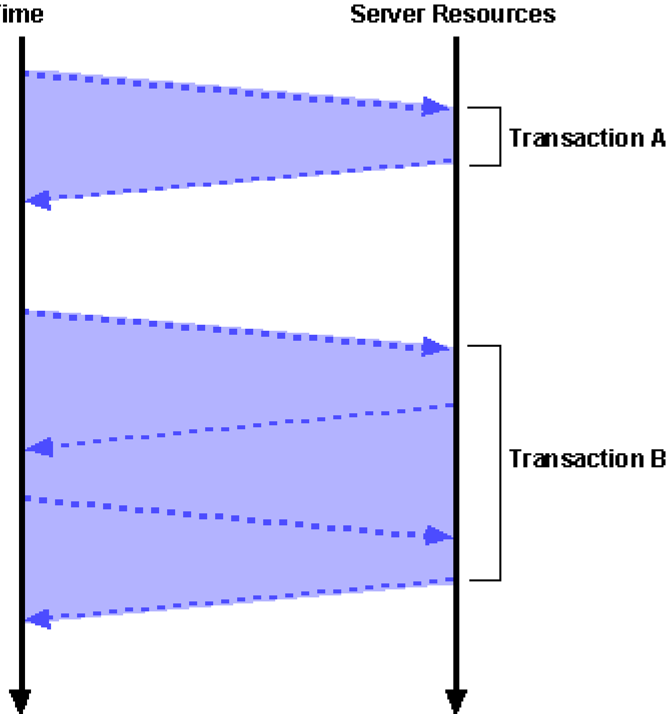
Why does MoveMoney outperform Stateful MoveMoney?

In the previous section, you saw that the time per transaction in **MoveMoney** and **Stateful MoveMoney** using a single Sample Bank client is nearly the same. However, as the number of concurrent transactions increases, **MoveMoney** begins to outperform **Stateful MoveMoney** significantly . At first glance, the code doesn't seem to account for the lag.

Class member variables for each account must be set before calling StatefulPerform, whereas Perform passes the account numbers by value through function parameters. The call to return the value of the account number in the MoveMoney object isn't an intensive operation. So what explains the performance degradation?

The reason is that Microsoft Transaction Server cannot commit transactions until it completes a method call. To maintain internal state, additional method calls are made on the MoveMoney object, thereby delaying the object from completing its work. This delay may cause server resources, such as database connections, to be held longer, therefore decreasing the amount of resources available for other clients. In other words, the application won't scale well.

The following diagram illustrates this point. The arrow on the left indicates time, which translates into performance. The arrow on the right indicates the server resources consumed, which translates into throughput. Transaction A represents a call made to stateless objects. On return from the method call, Transaction Server determines that the transaction can be committed, allowing the object to release its resources and be deactivated. On the other hand, Transaction B holds state between method calls, which increases the time that the server holds onto resources for that transaction. As the number of clients increases, so does the time required for transactions to be completed.



Another Pitfall When Using Stateful Objects

Examine the following excerpt from the Sample Bank client code (some code has been omitted for clarity).

```
For i = 1 To nTrans
   .
   .
   .
   obj.PrimeAccount = PrimeAcct
   obj.SecondAccount = lSecondAcct
   Res = obj.StatefulPerform(CLng(Amount), TranType
   .
   .
   .
Next i
```

Because the account numbers don't change, you might be inclined to rearrange the code as follows:

```
obj.PrimeAccount = PrimeAcct
obj.SecondAccount = lSecondAcct

For i = 1 To nTrans
   Res = obj.StatefulPerform(CLng(Amount), TranType
Next i
```

If you modify the code and then run the Sample Bank client for multiple transactions, the application fails on the second transaction. Why?

The answer is subtle. MoveMoney uses **SetComplete** to notify Transaction Server that it has completed its work. At this point, the MoveMoney object is deactivated. In the process of deactivation, all of the object's member variables are reinitialized. The next call to MoveMoney causes just-in-time activation. The activated object is now in its initial state, meaning the values of PrimeAccountNo and SecondAccountNo are both zero. Thus, the next call to StatefulPerform fails because of an invalid account number.

This is yet another reason to be careful when maintaining state in objects. Clients of application objects must be aware of how an object uses **SetComplete** to ensure that any state the object maintains won't be needed after the object undergoes just-in-time activation.

See Also

Transactions, Deactivating Objects, Context Objects, Stateful Components, **ObjectContext** object, **SetComplete** method

Multiple Transactions

This section explains the benefits of distributing work among multiple transactions.

- **Scenario: Storing Receipt Numbers in a Database**
  Add the UpdateReceipt component, which stores a maximum receipt number in a database and runs in a new transaction.
- **Creating the UpdateReceipt Component**
  Create the UpdateReceipt component, and modify the Receipt component to use UpdateReceipt.
- **Application Design Notes: Using Separate Transactions**
  Learn why this scenario requires separate transactions.

See Also

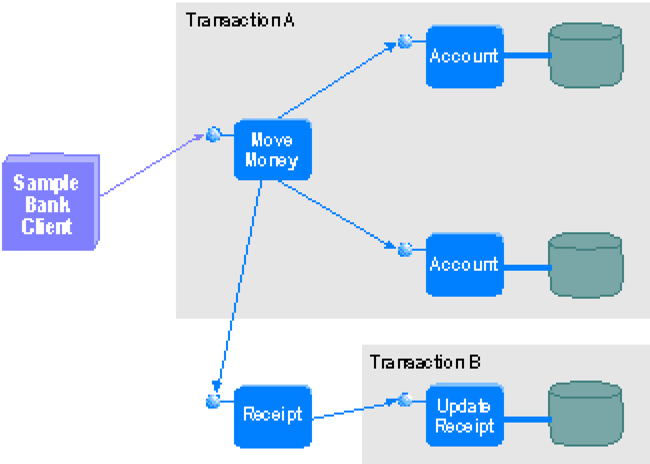
Transactions, Transaction Attributes, Activities

Scenario: Storing Receipt Numbers in a Database

In Sharing State, you added the Receipt component, which assigns a unique receipt number to each monetary transaction. The Shared Property Manager maintains these values, so they exist for the duration of the server process. In this section, you will add code to store a maximum receipt number in a database. Storing them makes receipt numbers unique beyond the life of the server process.

You will create the UpdateReceipt component, which stores a maximum receipt number in a database. When this maximum is reached, which happens on every one-hundredth transaction, UpdateReceipt adds 100 to the maximum receipt value and updates the database.

You will install the UpdateReceipt component so that it creates a new transaction, separate from those of the Account objects. The section will then discuss the advantages of using multiple transactions in this scenario. The application looks like the following figure (the Shared Property Manager and its associated objects are omitted for clarity):



See Also

Transactions, Transaction Attributes, Activities, Sharing State

Creating the UpdateReceipt Component

To implement the scenario for this section, you will build the UpdateReceipt component. You will also modify the Receipt component's Update method to use UpdateReceipt. Update adds 100 to the maximum receipt value stored in the database.

▶ Click here to see the Update method

You also need to add code to the GetNextReciept method of the Receipt component to check whether the maximum receipt value has been reached. If so, the Update method is called.

▶ Click here to see the GetNextReceipt method

▶ **To create the UpdateReceipt component**

1. Open the \Mts\Samples\Account.VB\Step7\Account.vbp project.
2. Build the component as a dynamic-link library (DLL) and save it as \Mts\Samples\Account.VB\Step7\VBAcct.dll.

By adding a new class module, you add a new COM component to this DLL. Therefore, you need to delete the existing components in the Microsoft Transaction Server Explorer and then install the new components.

▶ **To reinstall your components**

1. Remove the Account, MoveMoney, CreateTable, and Receipt components from the Transaction Server Explorer.

▶ How?

2. Add the new components. Use the DLL you created in \Mts\Samples\Account.VB\Step7\VBAcct.dll.

▶ How?

▶ **To set the transaction attributes for your components**

1. For the Account and MoveMoney components, set the transaction attribute to **Requires a transaction**.

▶ How?

2. For the Receipt component, set the transaction attribute to **Does not support transactions**. This is the default value.
3. For the CreateTable and UpdateReceipt components, set the transaction attribute to **Requires a new transaction**.

The code you added here is similar to the code you added in "Building Transactional Components." However, choosing **Requires a new transaction** causes the UpdateReceipt component to run in a new transaction. The next section discusses how this affects application behavior.

See Also

Transactions, Transaction Attributes

Application Design Notes: Using Separate Transactions

In Building Transactional Components, you saw the benefits of composing work under a transaction. The scenario in this section demonstrates a case in which using multiple transactions within an activity is required.

The major functional change in this scenario is the addition of the UpdateReceipt component, which makes the maximum receipt number durable by storing it in a database. As in Sharing State, the Shared Property Manager stores the receipt number. On every 100 transactions, the value in the database is incremented by 100. This dispenses a block of receipt numbers that are assigned to the next 100 transactions.

The UpdateReceipt component has a transaction attribute of **Requires a new transaction**. This guarantees that UpdateReceipt's work happens in a separate transaction. Thus, there is no connection between the success or failure of Account's work and UpdateReceipt's work.

This might appear to lower the fault tolerance of the application. For example, if the Account object aborts the transaction, a receipt number is still assigned. Therefore, skips in the receipt number sequence are possible. However, the application doesn't really need consecutively increasing receipt numbers—it just requires that there be no duplicate receipts. In this scenario, it's more important for the monetary transaction to be completed properly. Furthermore, requesting an update on every one-hundredth transaction improves performance by conserving calls to the database.

Composing both database updates under a single transaction would reduce the application's scalability. Even though UpdateReceipt is a simple update, it would consume more server resources because the database connection would have to be maintained until the Account object has completed its work. Thus, locks would be held longer than necessary, preventing other clients from writing to the database. Only when all work has been completed could these resources be freed.

See Also

Transactions, Transaction Attributes

Secured Components

This chapter shows how to use Microsoft Transaction Server's security features to restrict the use of application features to designated users.

- **Scenario: Adding Role Checking to the MoveMoney and Account Components**

  Add role checking to the MoveMoney and Account components to limit the transaction amount for designated users.

- **Using IsCallerInRole in the MoveMoney and Account Components**

  Use the **IsCallerInRole** method in the MoveMoney and Account components to verify that the user running the Bank client is a manager.

- **Application Design Notes: Using Roles**

  Learn how roles are useful in building secured components and how security boundaries are determined.

See Also

Programmatic Security

Scenario: Adding Role Checking to the MoveMoney and Account Components

For this scenario, you will limit which users have the ability to perform transactions of more than $500. You will add code to the MoveMoney and Account components that checks to see if the user of the Bank client is a manager. This is accomplished by defining a Manager role. Roles provide the flexibility a developer needs to build secured components without tying the implementation to a specific deployment domain.

See Also

Programmatic Security, Application Design Notes: Using Roles

Using IsCallerInRole in the MoveMoney and Account Components

You will add the **IsCallerInRole** method to the MoveMoney and Account components to verify that the user running the Bank client is a manager. This additional code is the same for both components. You must modify both components because clicking **Account** in the Bank client doesn't use the MoveMoney component when the Sample Bank application runs.

To use IsCallerInRole in the MoveMoney and Account components

1. Open the \Mts\Samples\Account.VB\Step8\Account.vbp project.

Click here to see the modified MoveMoney component

Click here to see the modified Account component

2. Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step8\VBAcct.dll.

IsCallerInRole is a method on an object's context. **IsCallerInRole** returns TRUE if the direct caller of that object is assigned to a given role. You will use **IsCallerInRole** in the MoveMoney and Account components to verify if the caller of an object in this case the user running the Bank client is a manager.

```
If (lngAmount > 500 Or lngAmount < -500) Then
    If Not GetObjectContext.IsCallerInRole("Managers") Then
        Err.Raise Number:=APP_ERROR, _
        Description:="Need 'Managers' role for amounts over $500"
    End If
End If
```

Before you can use the new MoveMoney and Account components, you must create the role. The Manager role must exist before the call to **IsCallerInRole**; otherwise, you will get an error.

Note that the source code is bound to a role name scoped to a package. This creates a dependancy between the source and the package definition that must be considered when making modifications to a Package's security configuration, such as deleting a role.

To define a role for the Sample Bank package

1. Start the Microsoft Transaction Server Explorer.

   If you are currently running Sample Bank, you must shut down the associated server process to change security properties.

How?

2. Create a role named Manager.

How?

3. Assign users to the role. If you have access to more than one Windows NT account, you may want to exclude some user accounts from the Manager role to see the role checking in effect.

How?

Run the Bank client. If you are logged on as a user in the Manager role, you will be able to perform transactions of any amount. However, if you are logged on as a user who isn't in the Manager role, you will get a warning message when attempting a transaction of more than $500. The transaction will then abort. If you don't have access to more than one account, try removing your user account from the role to see the role checking enforced.

How?

See Also

Programmatic Security, Enabling MTS Package Security, Application Design Notes: Using Roles, **IsCallerInRole** method

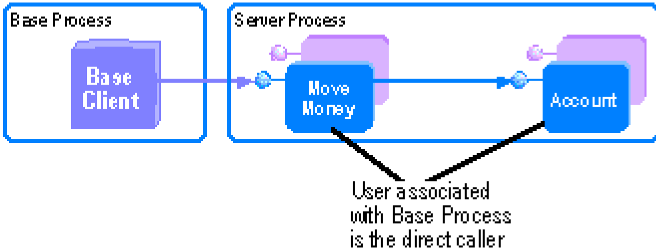
Application Design Notes: Using Roles

This section explains how roles are useful in building secured components. It also discusses how deployment configuration determines security boundaries.

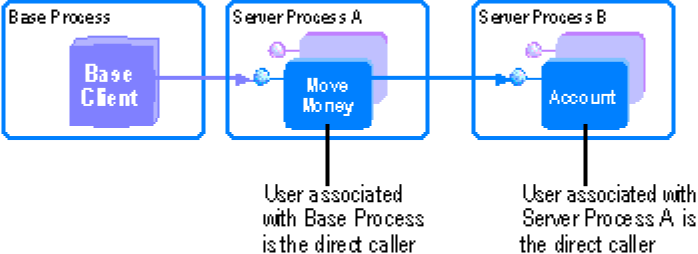
Roles

A role is a symbolic name that defines a group of users for a package of components. Roles extend Windows NT security to allow a developer to build secured components in a distributed application. In this scenario, the Manager role is defined at development time, but not yet bound to specific users. For each deployment of the application, the administrator can then assign the users and groups to a role in order to customize the application for his or her business.

In this scenario, role checking is done by the MoveMoney and Account objects. When both MoveMoney and Account objects are used, the second check (on the Account object) is redundant. However, it yields the same result, because **IsCallerInRole** applies to the direct caller, and both the MoveMoney and Account objects run in the same server process.



If you place the MoveMoney and Account components into separate packages, the components run in separate server processes. In this scenario, calling **IsCallerInRole** on the Account object context would check if the MoveMoney object's associated server process is running in the Manager role. MoveMoney is now the direct caller because a process boundary has been crossed.



The Account object runs under a package identity that gives that process full access to the bank account database. Account objects have the authority to update any account for any amount. Roles provide a means of permitting and denying access to objects. Once this permission is granted, the client, in effect, has the same access rights as the server process.

When you configured the package, you chose a package identity of **Interactive User**. In a real-world scenario, packages are more likely to run as a specific user, such as SampleBank, which has access rights to the database.

Returning to the scenario where you split the MoveMoney and Account components into separate packages, running as the SampleBank user solves the role checking problem. Adding the SampleBank user to the Manager role would allow the second **IsCallerInRole** check (on the Account object) to always succeed.

Security Boundaries Are Process-Wide

Transaction Server security is enabled only within a server process. Because the MoveMoney component is configured to run within a server process, role checking is enabled.

If you configure the Sample Bank components to run in-process, role checking would be disabled. In this case, **IsCallerInRole** always returns TRUE, which means the direct caller would always pass the authorization check.

You could use the **IsSecurityEnabled** method to check if Transaction Server security can be used. **IsSecurityEnabled** returns FALSE when the object runs in-process. Using **IsSecurityEnabled**, you could rewrite the role-checking code to disable transactions when objects aren't running in a secured environment.

In-process components share the same level of trust as the base client. Because of this, it isn't recommended that you deploy your secured components to be loaded in-process with their clients.

See Also

Programmatic Security, Enabling MTS Package Security, **IsCallerInRole** method, **IsSecurityEnabled** method

MTS Reference

Visual Basic

Functions
Methods
Objects
Properties
Language Summary

Visual C++

Functions
Interfaces
Methods
Language Summary

Visual J++

Interfaces
Methods
Language Summary

Functions (Visual Basic)

GetObjectContext

SafeRef

Objects (Visual Basic)

You use the following object in base clients:

TransactionContext

You use the following objects in components:

ObjectContext

SecurityProperty

SharedProperty

SharedPropertyGroup

SharedPropertyGroupManager

Methods (Visual Basic)

Abort

Activate

CanBePooled

Commit

Count

CreateInstance

CreateInstance (Transaction Context)

CreatePropertyGroup

CreateProperty

CreatePropertyByPosition

Deactivate

DisableCommit

EnableCommit

GetDirectCallerName

GetDirectCreatorName

GetOriginalCallerName

GetOriginalCreatorName

IsCallerInRole

IsInTransaction

IsSecurityEnabled

Item

SetAbort

SetComplete

Properties (Visual Basic)

Group

Property

PropertyByPosition

Value

Language Summary (Visual Basic)

Abort method

Activate method

CanBePooled method

Commit method

Count method

CreateInstance method

CreateInstance method (Transaction Context)

CreateProperty method

CreatePropertyByPosition method

CreatePropertyGroup method

Deactivate method

DisableCommit method

EnableCommit method

GetDirectCallerName method

GetDirectCreatorName method

GetObjectContext function

GetOriginalCallerName method

GetOriginalCreatorName method

Group property

IsCallerInRole method

IsInTransaction method

IsSecurityEnabled method

Item method

ObjectContext object

Property property

PropertyByPosition property

SafeRef function

SecurityProperty object

SetAbort method

SetComplete method

SharedProperty object

SharedPropertyGroup object

SharedPropertyGroupManager object

TransactionContext object

Value property

Functions (Visual C++)

GetObjectContext

SafeRef

Interfaces (Visual C++)

You use the following object in base clients:

ITransactionContext

You use the following objects in components:

IGetContextProperties

IObjectContext

IObjectContextActivity

IObjectControl

ISecurityProperty

ISharedProperty

ISharedPropertyGroup

ISharedPropertyGroupManager

Methods (Visual C++)

Abort

Activate

CanBePooled

Commit

Count

CreateInstance

CreateInstance (Transaction Context)

CreatePropertyGroup

CreateProperty

CreatePropertyByPosition

Deactivate

DisableCommit

EnableCommit

EnumNames

get__NewEnum

get_Group

get_Property

get_PropertyByPosition

get_Value

GetActivityId

GetDirectCallerSID

GetDirectCreatorSID

GetOriginalCallerSID

GetOriginalCreatorSID

GetProperty

IsCallerInRole

IsInTransaction

IsSecurityEnabled

put_Value

ReleaseSID

SetAbort

SetComplete

Language Summary (Visual C++)

Abort method

Activate method

CanBePooled method

Commit method

Count method

CreateInstance method

CreateInstance method (Transaction Context)

CreateProperty method

CreatePropertyByPosition method

CreatePropertyGroup method

Deactivate method

DisableCommit method

EnableCommit method

EnumNames method

get__NewEnum method

get_Group method

get_Property method

get_PropertyByPosition method

get_Value method

GetActivityId method

GetDirectCallerSID method

GetDirectCreatorSID method

GetObjectContext function

GetOriginalCallerSID method

GetOriginalCreatorSID method

GetProperty method

IGetContextProperties interface

IObjectContext interface

IObjectContextActivity interface

IObjectControl interface

IsCallerInRole method

ISecurityProperty interface

ISharedProperty interface

ISharedPropertyGroup interface

ISharedPropertyGroupManager interface

IsInTransaction method

IsSecurityEnabled method

ITransactionContextEx interface

put_Value method

ReleaseSID method

SafeRef function

SetAbort method

SetComplete method

Interfaces (Visual J++)

You use the following object in base clients:

ITransactionContext

You use the following objects in components:

IGetContextProperties

IObjectContext

IObjectControl

ISharedProperty

ISharedPropertyGroup

ISharedPropertyGroupManager

Methods (Visual J++)

Abort

Activate

CanBePooled

Commit

Count

CreateInstance

CreateInstance (Transaction Context)

CreatePropertyGroup

CreateProperty

CreatePropertyByPosition

Deactivate

DisableCommit

EnableCommit

EnumNames

get_NewEnum

getGroup

getProperty

GetProperty

getPropertyByPosition

getValue

IsCallerInRole

IsInTransaction

IsSecurityEnabled

MTx.GetObjectContext

MTx.SafeRef

putValue

SetAbort

SetComplete

Language Summary (Visual J++)

Abort method

Activate method

CanBePooled method

Commit method

Count method

CreateInstance method

CreateInstance method (Transaction Context)

CreateProperty method

CreatePropertyByPosition method

CreatePropertyGroup method

Deactivate method

DisableCommit method

EnableCommit method

EnumNames method

get_NewEnum method

getGroup method

getProperty method

GetProperty method

getPropertyByPosition method

getValue method

IObjectContext interface

IObjectControl interface

IsCallerInRole method

ISharedProperty interface

ISharedPropertyGroup interface

ISharedPropertyGroupManager interface

IsInTransaction method

IsSecurityEnabled method

ITransactionContextEx interface

MTx.GetObjectContext method

MTx.SafeRef method

putValue method

SetAbort method

SetComplete method

IGetContextProperties Interface

The **IGetContextProperties** interface provides access to context object properties.

Remarks

The header file for the **IGetContextProperties** interface is mtx.h.

The **IGetContextProperties** interface exposes the following methods.

Method	Description
Count	Returns the number of properties associated with the context object.
EnumNames	Returns a reference to an enumerator that you can use to iterate through all the context object properties.
GetProperty	Returns a context object property.

See Also

Context Objects

IGetContextProperties::Count Method

Returns the number of context object properties.

Provided By

IGetContextProperties

HRESULT IGetContextProperties::Count (
    long * *plCount***);**

Parameters

plCount
  [out] The number of properties.

Return Values

S_OK
  A valid count is returned in the *plCount* parameter.

E_INVALIDARG
  The valid count could not be returned.

Example

IGetContextProperties::EnumNames Method

Returns a reference to an enumerator that you can use to iterate through all the context object properties.

Provided By

IGetContextProperties

HRESULT IGetContextProperties::EnumNames (
  IEnumNames ** *ppenum***);**

Parameters

ppenum
  [out] A reference to the **IEnumNames** interface on a new enumerator object that you can use to iterate through all the context object properties.

Return Values

S_OK
  A reference to the requested enumerator is returned in the *ppenum* parameter.

E_INVALIDARG
  The requested enumerator could not be returned.

Remarks

You use the **EnumNames** method to obtain a reference to an enumerator object. The returned **IEnumNames** interface exposes several methods you can use to iterate through a list of BSTRs representing context object properties. Once you have a name, you can use the **GetProperty** method to obtain a reference to the context object property it represents. See the COM documentation for information on enumerators.

As with any COM object, you must release an enumerator object when you're finished using it.

Example

IGetContextProperties::GetProperty Method

Returns a context object property.

Provided By

IGetContextProperties

HRESULT IGetContextProperties::GetProperty (
    BSTR *name*
    VARIANT * *pProperty***);**

Parameters

name
  [in] The name of the context object property to be retrieved.

pProperty
  [out] The returned pointer to the property.

Return Values

S_OK
  The property was returned successfully.

S_FALSE
  The property was not found.

E_INVALIDARG
  The *name* parameter was invalid.

Remarks

You can use **GetProperty** to retrieve the following Microsoft Internet Information Server (IIS) intrinsic objects:

- Request
- Response
- Server
- Application
- Session

For more information, see the IIS documentation.

To retrieve an IIS object, call **QueryInterface** using the VT_DISPATCH member of the returned VARIANT for the interface to the IIS object (for example **IReponse** for the Response object).

Note Context properties are not available in MTS 1.0 server processes.

Example

Count, EnumNames, GetProperty Methods Example

```
#include "stdafx.h"
#include <initguid.h>
#include "asptlb.h"
#include "mtx.h"

HRESULT hr = NOERROR;

// Get the context object
CComPtr<IObjectContext> pObjectContext;
hr = GetObjectContext(&pObjectContext);

// Get the Response object
CComVariant v;
CComBSTR bstr(L"Response");
CComQIPtr<IGetContextProperties,   &IID_IGetContextProperties>
pProps(pObjectContext);
hr = pProps->GetProperty(bstr, &v);
CComPtr<IDispatch> pDisp;
pDisp = V_DISPATCH(&v);
CComQIPtr<IResponse, &IID_IResponse> pResponse(pDisp);

// Print number of properties
long lCount;
hr = pProps->Count(&lCount);
bstr = L"<p>Number of properties: ";
CComBSTR bstrCount;
VarBstrFromI4(lCount, 0, 0, &bstrCount);
bstr.Append(bstrCount);
bstr.Append(L"</p>");
v = bstr;
hr = pResponse->Write(v);

// Iterate over properties collection and print the
// names of the properties
CComPtr<IEnumNames> pEnum;
CComBSTR bstrTemp;
pProps->EnumNames(&pEnum);
for ( int i = 0; i < lCount; ++i )
{
   pEnum->Next(1, &bstr, NULL);
   bstrTemp = L"<p>";
   bstrTemp.Append(bstr);
   bstrTemp.Append(L"</p>");
   v = bstrTemp;
   hr = pResponse->Write(v);
}
```

IGetContextProperties Interface

The **IGetContextProperties** interface provides access to context object properties.

Remarks

The **IGetContextProperties** interface is declared in the package `com.ms.mtx.`

The **IGetContextProperties** interface exposes the following methods.

Method	Description
Count	Returns the number of properties associated with the context object.
EnumNames	Returns a reference to an enumerator that you can use to iterate through all the context object properties.
GetProperty	Returns a context object property.

See Also

Context Objects

IGetContextProperties.Count Method

Returns the number of context object properties.

Provided By

IGetContextProperties

int Count( );

Return Value

The number of properties.

Example

IGetContextProperties.EnumNames Method

Returns a reference to an enumerator that you can use to iterate through all the context object properties.

Provided By

IGetContextProperties

IEnumNames EnumNames ( );

Return Value

A reference to the **IEnumNames** interface on a new enumerator object that you can use to iterate through the list of all the shared property groups in the process.

Remarks

You use the **EnumNames** method to obtain a reference to an enumerator object. The returned **IEnumNames** interface exposes several methods you can use to iterate through a list of string expressions representing context object properties. Once you have a name, you can use the **GetProperty** method to obtain a reference to the context object property it represents. See the COM documentation for information on enumerators.

Example

IGetContextProperties.GetProperty Method

Returns a context object property.

Provided By

IGetContextProperties

Variant GetProperty(
    String *name***);**

Parameters

name
  [in] The name of the context object property to be retrieved.

Return Value

The requested context object property.

Remarks

You can use **GetProperty** to retrieve the following Microsoft Internet Information Server (IIS) intrinsic objects:

- Request
- Response
- Server
- Application
- Session

For more information, see the IIS documentation.

To retrieve an IIS object, use the **getDispatch** method of returned Variant and cast this value to the interface to the IIS object (for example **IReponse** for the Response object).

Note   Context properties are not available in MTS 1.0 server processes.

Example

Count, EnumNames, GetProperty Methods Example

```
import com.ms.com.*;
import com.ms.mtx.*;
import asp.*;

// Get the context object
IGetContextProperties icp;
IResponse iresp;
Variant av = new Variant();
icp = (IGetContextProperties)MTx.GetObjectContext();

// Get the Response object
av = icp.GetProperty("Response");
iresp = (IResponse) av.getDispatch();
av.VariantClear();

// Print number of properties
String str;
int pc;
pc = icp.Count();
str = "<p>Number of properties: " + pc + "</p>";
av.putString(str);
iresp.Write (av);
av.VariantClear();

// Iterate over properties collection and print the
// names of the properties
IEnumNames iEnum = null;
int howmany = 1;
String[] names = new String[1];
int fetched;
iEnum = icp.EnumNames();
for ( int i = 0; i < pc; ++i )
{
   fetched = iEnum.Next( howmany, names);
   str = "<p>" + names[0] + "</p>";
   av.putString(str);
   iresp.Write (av);
   av.VariantClear();
}
```

ObjectControl Object

You implement the **ObjectControl** interface when you want to define context-specific initialization and cleanup procedures for your MTS objects and specify whether or not the objects can be recycled. Implementing the **ObjectControl** interface is optional.

Remarks

To use the **ObjectControl** object, you must set a reference to the Microsoft Transaction Server Type Library (MTxAS.dll).

If you implement the **ObjectControl** interface in your component, the MTS run-time environment automatically calls the **ObjectControl** methods on your objects at the appropriate times.

When an object supports the **ObjectControl** interface, MTS calls its **Activate** method once for each time the object is activated. The **Activate** method is called before any of the object's other methods are called. You can use this method to perform any context-specific initialization an object may require.

MTS calls the object's **Deactivate** method each time the object is deactivated. This can be the result of the object returning from a method in which it calls **SetComplete** or **SetAbort**, or due to the root of the object's transaction, causing the transaction to complete. You use the **Deactivate** method to clean up state that you initialized in the **Activate** method.

After calling the **Deactivate** method, the MTS run-time environment calls the **CanBePooled** method. If this method returns **True**, the deactivated object is placed in an object pool for reuse. If the **CanBePooled** method returns **False**, the object is released in the usual way.

Note On systems that don't support object pooling, the value returned by this method is ignored.

The **ObjectControl** interface is not accessible to an object's clients or to the object itself. Only the MTS run-time environment can invoke the **ObjectControl** methods.

The **ObjectControl** interface provides the following methods.

Method	Description
Activate	Allows an object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.
CanBePooled	Allows an object to notify the MTS run-time environment of whether it can be pooled for reuse. Return **True** if you want instances of this component to be pooled, or **False** if not.
Deactivate	Allows an object to perform whatever cleanup is necessary before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

See Also

Deactivating Objects, Object Pooling and Recycling

Activate Method

Implementing this method allows an MTS object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.

Applies To

ObjectContext Object

Syntax

*objectcontrol.***Activate**

The *objectcontrol* placeholder represents is an object variable that evaluates to an **ObjectControl** object.

Remarks

To use the **ObjectControl** object, you must set a reference to the Microsoft Transaction Server Type Library (MTxAS.dll).

Whenever a client calls an MTS object that isn't already activated, the MTS run-time environment automatically activates the object. This is called just-in-time activation. For components that support the **ObjectControl** interface, MTS invokes the object's **Activate** method before passing the client's method call on to the object. This allows objects to do context-specific initialization.

If you need to perform any initialization that involves the **ObjectContext**, you should implement the **Activate** method and place all your context-specific initialization procedures there.

For example, you can use the **Activate** method to obtain a reference to an object's context and store it in a member variable. Then the context is available to any method that requires it, and you don't have to acquire a new one and then release it every time you use it. Once you have a reference to the object's context, you can use the **IObjectContext** methods to check whether security is enabled, whether the object is executing in a transaction, or whether the caller is in a particular role.

If you're enabling object recycling (returning **True** from the **CanBePooled** method), the **Activate** method must be able to handle both newly created and recycled objects. When the **Activate** method returns, there should be no distinguishable difference between a new object and a recycled one.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

Activate Method Example

```
Implements ObjectControl
Dim context As ObjectContext
Option Explicit

Private Sub ObjectControl_Activate()
   ' Get a reference to the object's context here,
   ' so it can be used by any method that may be
   ' called during this activation of the object.
   Set context = GetObjectContext()
End Sub
```

CanBePooled Method

Implementing this method allows an MTS object to notify the MTS run-time environment of whether it can be pooled for reuse. Return **True** if you want the object to be pooled for reuse, or **False** if not.

Applies To

ObjectContext Object

Syntax

*objectcontrol.***CanBePooled (True | False**)

The *objectcontrol* placeholder represents is an object variable that evaluates to an **ObjectControl** object.

Return Values

True
Notifies the MTS run-time environment that it can pool this object on deactivation for later reuse.

False
Notifies the MTS run-time environment that it should not pool this object on deactivation, but should release its last reference to the object so that the object will be destroyed.

Remarks

To use the **ObjectControl** object, you must set a reference to the Microsoft Transaction Server Type Library (MTxAS.dll).

When an object returns **True** from the **CanBePooled**method, it indicates to the MTS run-time environment that it can be added to an object pool after deactivation rather than being destroyed. Whenever an instance is required, one is drawn from the pool rather than created.

The way recycling works is that an object cleans itself up in its **Deactivate** method and is returned to an object pool. Later, when an instance of the same component is needed, the cleaned up object can be reused. For this to work, an object must be accessible on different threads each time it's activated. Recycling isn't possible under the apartment threading model because, in that model, although an object can be instantiated on any thread, it can only be used by the thread on which it was instantiated. If you want instances of a component to be recyclable, you should register the component with the **ThreadingModel** Registry value set to **Both**. This indicates to MTS that the component's objects can be called from different threads.

In MTS, these objects will run under the apartment threading model and won't be recycled even if they return **True** from the **CanBePooled** method. However, if you configure a component to support both threading models, the component will run under the current version of MTS and will also be able to take advantage of recycling as soon as it becomes available, without any changes to the code.

Deciding whether to enable recycling is a matter of weighing the costs and benefits. Recycling requires writing additional code, and there's a risk that some state may be inadvertently retained from one activation to the next. When you allow objects to be pooled, you have to be very careful in your **Activate** and **Deactivate** methods to ensure that a recycled object is always restored to a state that's equivalent to the state of a newly created object. Another consideration to take into account is the amount of resources required to maintain an object pool. Objects that hold a lot of resources can be expensive to pool. However, in certain situations, recycling can be extremely efficient, resulting in improved performance and increased scalability. The trade-off is between the cost of holding onto resources while objects are pooled (and inactive) versus the cost of creating and destroying the resources.

It's usually best to enable recycling for objects that cost more to create than they cost to reinitialize.

For example, if a component contains a complex structure, and that structure can be reused, it could save a lot of time if the structure didn't have to be recreated every time an instance of the component was activated. This is a case in which you might want to enable recycling, which you would do by returning **True** from the **CanBePooled** method.

The **Activate** method is called whether a new instance is created or a recycled instance is drawn from the pool. Similarly, the **Deactivate** method is called every time the object is deactivated, whether it's being destroyed or returned to the pool for recycling.

So, in this example, you'd use the object's **Activate** method to initialize, or reinitialize, the structure that's being reused, and you'd use the **Deactivate** method to restore the object to a state that the **Activate** method can handle. (The **Activate** method must be able to handle both new objects and reused objects drawn from the pool.) This combined use of the **Activate**, **Deactivate**, and **CanBePooled** methods eliminates the need to recreate reusable resources every time an instance is activated.

For some objects, recycling isn't efficient. For example, if an object acquires a lot of state during its lifetime that isn't reusable, and has little to do during its construction, it's usually cheaper to create a new instance whenever one is needed. In that case, you would return **False** from the **CanBePooled** method.

Note Returning **True** from the **CanBePooled** method doesn't guarantee that objects will be recycled; it only gives the MTS run-time environment permission to recycle them. On systems that don't support object pooling, a return value of **True** is ignored. Returning **False** from the **CanBePooled** method guarantees that instances of a component aren't recycled.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

CanBePooled Method Example

```
Implements ObjectControl
Dim context As ObjectContext
Option Explicit

Private Function ObjectControl_CanBePooled() As Boolean
   ' This object should not be recycled,
   ' so return false.
   ObjectControl_CanBePooled = False
End Function
```

Deactivate Method

Implementing this method allows an MTS object to perform any cleanup required before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

Applies To

ObjectContext Object

Syntax

objectcontrol.**Deactivate**

The *objectcontrol* placeholder represents is an object variable that evaluates to an **ObjectControl** object.

Remarks

To use the **ObjectControl** object, you must set a reference to the Microsoft Transaction Server Type Library (MTxAS.dll).

The MTS run-time environment calls the **Deactivate** method whenever an object that supports the **ObjectControl** interface is deactivated. An object is deactivated when it returns from a method in which it called **SetComplete** or **SetAbort**, when the transaction in which it executed is committed or aborted, or when the last client to hold a reference on it releases its reference.

If the component supports recycling (returns **True** from the **CanBePooled** method), you should use the **Deactivate** method to reset the object's state to the state in which the **Activate** method expects to find it. You can also use the **Deactivate** method to release the object's **ObjectContext** or to do other context-specific cleanup. Even if an object supports recycling, it can be beneficial to release certain reusable resources in its **Deactivate** method. For example, ODBC provides its own connection pooling. It's more efficient to pool a database connection in a general connection pool where it can be used by other objects than it is to keep it tied to a specific object in an object pool.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

Deactivate Method Example

```
Implements ObjectControl
Dim objcontext As ObjectContext
Option Explicit

Private Sub ObjectControl_Deactivate()
   ' Perform any necessary cleanup here.
   Set objcontext = Nothing
End Sub
```

IObjectControl Interface

You implement the **IObjectControl** interface when you want to define context-specific initialization and cleanup procedures for your MTS objects and specify whether or not the objects can be recycled. Implementing the **IObjectControl** interface is optional.

Remarks

The header file for the **IObjectControl** interface is mtx.h.

If you implement the **IObjectControl** interface in your component, the MTS run-time environment automatically calls the **IObjectControl** methods on your objects at the appropriate times.

When an object supports the **IObjectControl** interface, MTS calls its **Activate** method once for each time the object is activated. The **Activate** method is called before any of the object's other methods are called. You can use this method to perform any context-specific initialization an object may require.

Note An object's context isn't available during object construction (from the object's class factory), so context-specific initialization can't be performed in an object's constructor.

MTS calls the object's **Deactivate** method each time the object is deactivated. This can be the result of the object returning from a method in which it calls **SetComplete** or **SetAbort**, or due to the root of the object's transaction, causing the transaction to complete. You use the **Deactivate** method to clean up state that you initialized in the **Activate** method.

After calling the **Deactivate** method, the MTS run-time environment calls the **CanBePooled** method. If this method returns TRUE, the deactivated object is placed in an object pool for reuse. If the **CanBePooled** method returns FALSE, the object is released in the usual way and its destructor is invoked.

Note On systems that don't support object pooling, the value returned by this method is ignored.

The **IObjectControl** interface is not accessible to an object's clients or to the object itself. Only the MTS run-time environment can invoke the **IObjectControl** methods. If a client queries for the **IObjectControl** interface, **QueryInterface** will return E_NOINTERFACE.

The **IObjectControl** interface exposes the following methods.

Method	Description
Activate	Allows an object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.
CanBePooled	Allows an object to notify the MTS run-time environment of whether it can be pooled for reuse. Return TRUE if you want instances of this component to be pooled, or FALSE if not.
Deactivate	Allows an object to perform whatever cleanup is necessary before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl::Activate Method

Implementing this method allows an MTS object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.

Provided By

IObjectContext

HRESULT IObjectControl::Activate ( );

Return Values

S_OK
    The Activate method succeeded.

Failure HRESULT
    Any error code indicating why an object was unable to activate itself.

Remarks

Whenever a client calls an MTS object that isn't already activated, the MTS run-time environment automatically activates the object. This is called just-in-time activation. For components that support the **IObjectControl** interface, MTS invokes the object's **Activate** method before passing the client's method call on to the object. This allows objects to do context-specific initialization.

Because an object's context isn't available during object construction, you can't perform any initialization that involves the **ObjectContext**inside the constructor. Instead, you should implement the **Activate** method and place all your context-specific initialization procedures there.

For example, you can use the **Activate** method to obtain a reference to an object's context and store it in a member variable. Then the context reference is available to any method that requires it, and you don't have to acquire a new one and then release it every time you use it. Once you have a reference to the object's context, you can use the **IObjectContext** methods to check whether security is enabled, whether the object is executing in a transaction, or whether the caller is in a particular role. You can also use the **Activate** method to obtain a reference to the object's **SecurityProperty** and check the security ID of the object's creator before any methods are called.

If you're enabling object recycling (returning TRUE from the **CanBePooled** method), the **Activate** method must be able to handle both newly created and recycled objects. When the **Activate** method returns, there should be no distinguishable difference between a new object and a recycled one.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl::Activate Method Example

```
#include <mtx.h>

IObjectContext* m_pObjectContext;

HRESULT MyObject::Activate()
{
   // Get a reference to the object's context here,
   // so it can be used by any method that may be
   // called during this activation of the object.
   HRESULT hr = GetObjectContext(&m_pObjectContext);
   if (SUCCEEDED(hr))
      return S_OK;
   return hr;
}
```

IObjectControl::CanBePooled Method

Implementing this method allows an MTS object to notify the MTS run-time environment of whether it can be pooled for reuse. Return TRUE if you want the object to be pooled for reuse, or FALSE if not.

Provided By

IObjectContext

BOOL IObjectControl::CanBePooled ( );

Return Values

TRUE
  Notifies the MTS run-time environment that it can pool this object on deactivation for later reuse.

FALSE
  Notifies the MTS run-time environment that it should not pool this object on deactivation, but should release its last reference to the object so that the object will be destroyed.

Remarks

When an object returns TRUE from the **CanBePooled**method, it indicates to the MTS run-time environment that it can be added to an object pool after deactivation rather than being destroyed. Whenever an instance is required, one is drawn from the pool rather than created by the class factory.

The way recycling works is that an object cleans itself up in its **Deactivate** method and is returned to an object pool. Later, when an instance of the same component is needed, the cleaned up object can be reused. For this to work, an object must be accessible on different threads each time it's activated. Recycling isn't possible under the apartment threading model because, in that model, although an object can be instantiated on any thread, it can only be used by the thread on which it was instantiated. If you want instances of a component to be recyclable, you should register the component with the **ThreadingModel** Registry value set to **Both**. This indicates to MTS that the component's objects can be called from different threads.

In MTS, these objects will run under the apartment threading model and won't be recycled even if they return TRUE from the **CanBePooled** method. However, if you configure a component to support both threading models, the component will run under the current version of MTS and will also be able to take advantage of recycling as soon as it becomes available, without any changes to the code.

Deciding whether to enable recycling is a matter of weighing the costs and benefits. Recycling requires writing additional code, and there's a risk that some state may be inadvertently retained from one activation to the next. When you allow objects to be pooled, you have to be very careful in your **Activate** and **Deactivate** methods to ensure that a recycled object is always restored to a state that's equivalent to the state of a newly created object. Another consideration to take into account is the amount of resources required to maintain an object pool. Objects that hold a lot of resources can be expensive to pool. However, in certain situations, recycling can be extremely efficient, resulting in improved performance and increased scalability. The trade-off is between the cost of holding onto resources while objects are pooled (and inactive) versus the cost of creating and destroying the resources.

It's usually best to enable recycling for objects that cost more to create than they cost to reinitialize. For example, if a component contains a complex structure, and that structure can be reused, it could save a lot of time if the structure didn't have to be recreated every time an instance of the component was activated. This is a case in which you might want to enable recycling, which you would do by returning TRUE from the **CanBePooled** method.

You could still create the structure in the object's constructor and release it in the destructor. The constructor is only called once, when a new object is created by the class factory. When recycling is

enabled, that only happens when the object pool is empty. The **Activate** method, on the other hand, is called whether a new instance is created by the class factory or a recycled instance is drawn from the pool. Similarly, the object's destructor is only called when no further instances are needed, for example, when the server is shutting down. The **Deactivate** method is called every time the object is deactivated, whether it's being destroyed or returned to the pool for recycling.

So, in this example, you'd use the object's **Activate** method to initialize, or reinitialize, the structure that's being reused, and you'd use the **Deactivate** method to restore the object to a state that the **Activate** method can handle. (The **Activate** method must be able to handle both new objects created by the class factory and reused objects drawn from the pool.) This combined use of the **Activate**, **Deactivate**, and **CanBePooled** methods eliminates the need to recreate reusable resources every time an instance is activated.

For some objects, recycling isn't efficient. For example, if an object acquires a lot of state during its lifetime that isn't reusable, and has little to do during its construction, it's usually cheaper to create a new instance whenever one is needed. In that case, you would return FALSE from the **CanBePooled** method.

Note Returning TRUE from the **CanBePooled** method doesn't guarantee that objects will be recycled; it only gives the MTS run-time environment permission to recycle them. On systems that don't support object pooling, a return value of TRUE is ignored. Returning FALSE from the **CanBePooled** method guarantees that instances of a component aren't recycled.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl::CanBePooled Method Example

```
#include <mtx.h>

BOOL MyObject::CanBePooled()
{
   // This object should not be recycled,
   // so return false.
   return FALSE;
}
```

IObjectControl::Deactivate Method

Implementing this method allows an MTS object to perform any cleanup required before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

Provided By

IObjectContext

void IObjectControl::Deactivate ( );.

Remarks

The MTS run-time environment calls the **Deactivate** method whenever an object that supports the **IObjectControl** interface is deactivated. An object is deactivated when it returns from a method in which it called **SetComplete** or **SetAbort**, when the transaction in which it executed is committed or aborted, or when the last client to hold a reference on it releases its reference.

If the component supports recycling (returns TRUE from the **CanBePooled** method), you should use the **Deactivate** method to reset the object's state to the state in which the **Activate**method expects to find it. You can also use the **Deactivate** method to release the object's **ObjectContext** reference or to do other context-specific cleanup. You can't do this in the **Release** method or the destructor, because the **IObjectContext** interface isn't accessible from the destructor or any of the **IUnknown** methods. Even if an object supports recycling, it can be beneficial to release certain reusable resources in its **Deactivate** method. For example, ODBC provides its own connection pooling. It's more efficient to pool a database connection in a general connection pool where it can be used by other objects than it is to keep it tied to a specific object in an object pool.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl::Deactivate Method Example

```
#include <mtx.h>

void MyObject::Deactivate(void)
{
   // This object is no longer associated with this
   // context, so release the reference it acquired in
   // its Activate method.
   m_pObjectContext->Release();
}
```

IObjectControl Interface

You implement the **IObjectControl** interface when you want to define context-specific initialization and cleanup procedures for your MTS objects and specify whether or not the objects can be recycled. Implementing the **IObjectControl** interface is optional.

Remarks

The **IObjectControl** interface is declared in the package `com.ms.mtx.`

If you implement the **IObjectControl** interface in your component, the MTS run-time environment automatically calls the **IObjectControl** methods on your objects at the appropriate times.

When an object supports the **IObjectControl** interface, MTS calls its **Activate** method once for each time the object is activated. The **Activate** method is called before any of the object's other methods are called. You can use this method to perform any context-specific initialization an object may require.

Note An object's context isn't available during object construction (from the object's class factory), so context-specific initialization can't be performed in an object's constructor.

MTS calls the object's **Deactivate** method each time the object is deactivated. This can be the result of the object returning from a method in which it calls **SetComplete** or **SetAbort**, or due to the root of the object's transaction, causing the transaction to complete. You use the **Deactivate** method to clean up state that you initialized in the **Activate** method.

After calling the **Deactivate** method, the MTS run-time environment calls the **CanBePooled** method. If this method returns `true`, the deactivated object is placed in an object pool for reuse. If the **CanBePooled** method returns `false`, the object is released in the usual way.

Note On systems that don't support object pooling, the value returned by this method is ignored.

The **IObjectControl** interface is not accessible to an object's clients or to the object itself. Only the MTS run-time environment can invoke the **IObjectControl** methods.

The **IObjectControl** interface exposes the following methods.

Method	Description
Activate	Allows an object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.
CanBePooled	Allows an object to notify the MTS run-time environment of whether it can be pooled for reuse. Return `true` if you want instances of this component to be pooled, or `false` if not.
Deactivate	Allows an object to perform whatever cleanup is necessary before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl.Activate Method

Implementing this method allows an MTS object to perform context-specific initialization whenever it's activated. This method is called by the MTS run-time environment before any other methods are called on the object.

Provided By

IObjectContext

void Activate ( );

Remarks

Whenever a client calls an MTS object that isn't already activated, the MTS run-time environment automatically activates the object. This is called just-in-time activation. For components that support the **IObjectControl** interface, MTS invokes the object's **Activate** method before passing the client's method call on to the object. This allows objects to do context-specific initialization.

Because an object's context isn't available during object construction, you can't perform any initialization that involves the **ObjectContext**inside the constructor. Instead, you should implement the **Activate** method and place all your context-specific initialization procedures there.

For example, you can use the **Activate** method to obtain a reference to an object's context and store it in a member variable. Then the context reference is available to any method that requires it, and you don't have to acquire a new one and then release it every time you use it. Once you have a reference to the object's context, you can use the **IObjectContext** methods to check whether security is enabled, whether the object is executing in a transaction, or whether the caller is in a particular role.

If you're enabling object recycling (returning `true` from the **CanBePooled** method), the **Activate** method must be able to handle both newly created and recycled objects. When the **Activate** method returns, there should be no distinguishable difference between a new object and a recycled one.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl.Activate Method Example

```
import com.ms.mtx.*;

void Activate()
{
   // Get a reference to the object's context here,
   // so it can be used by any method that may be
   // called during this activation of the object.
   m_ObjectContext = MTx.GetObjectContext()
}
```

IObjectControl.CanBePooled Method

Implementing this method allows an MTS object to notify the MTS run-time environment of whether it can be pooled for reuse. Return `true` if you want the object to be pooled for reuse, or `false` if not.

Provided By

IObjectContext

boolean CanBePooled ( );

Return Values

`true`
: Notifies the MTS run-time environment that it can pool this object on deactivation for later reuse.

`false`
: Notifies the MTS run-time environment that it should not pool this object on deactivation, but should release its last reference to the object so that the object will be destroyed.

Remarks

When an object returns `true` from the **CanBePooled**method, it indicates to the MTS run-time environment that it can be added to an object pool after deactivation rather than being destroyed. Whenever an instance is required, one is drawn from the pool rather than created by the class factory.

The way recycling works is that an object cleans itself up in its **Deactivate** method and is returned to an object pool. Later, when an instance of the same component is needed, the cleaned up object can be reused. For this to work, an object must be accessible on different threads each time it's activated. Recycling isn't possible under the apartment threading model because, in that model, although an object can be instantiated on any thread, it can only be used by the thread on which it was instantiated. If you want instances of a component to be recyclable, you should register the component with the **ThreadingModel** Registry value set to **Both**. This indicates to MTS that the component's objects can be called from different threads.

In MTS, these objects will run under the apartment threading model and won't be recycled even if they return `true` from the **CanBePooled** method. However, if you configure a component to support both threading models, the component will run under the current version of MTS and will also be able to take advantage of recycling as soon as it becomes available, without any changes to the code.

Deciding whether to enable recycling is a matter of weighing the costs and benefits. Recycling requires writing additional code, and there's a risk that some state may be inadvertently retained from one activation to the next. When you allow objects to be pooled, you have to be very careful in your **Activate** and **Deactivate** methods to ensure that a recycled object is always restored to a state that's equivalent to the state of a newly created object. Another consideration to take into account is the amount of resources required to maintain an object pool. Objects that hold a lot of resources can be expensive to pool. However, in certain situations, recycling can be extremely efficient, resulting in improved performance and increased scalability. The trade-off is between the cost of holding onto resources while objects are pooled (and inactive) versus the cost of creating and destroying the resources.

It's usually best to enable recycling for objects that cost more to create than they cost to reinitialize. For example, if a component contains a complex structure, and that structure can be reused, it could save a lot of time if the structure didn't have to be recreated every time an instance of the component was activated. This is a case in which you might want to enable recycling, which you would do by returning `true` from the **CanBePooled** method.

You could still create the structure in the object's constructor because the constructor is only called once, when a new object is created by the class factory. When recycling is enabled, that only happens

when the object pool is empty. The **Activate** method, on the other hand, is called whether a new instance is created by the class factory or a recycled instance is drawn from the pool. Similarly, the **Deactivate** method is called every time the object is deactivated, whether it's being destroyed or returned to the pool for recycling.

So, in this example, you'd use the object's **Activate** method to initialize, or reinitialize, the structure that's being reused, and you'd use the **Deactivate** method to restore the object to a state that the **Activate** method can handle. (The **Activate** method must be able to handle both new objects created by the class factory and reused objects drawn from the pool.) This combined use of the **Activate**, **Deactivate**, and **CanBePooled** methods eliminates the need to recreate reusable resources every time an instance is activated.

For some objects, recycling isn't efficient. For example, if an object acquires a lot of state during its lifetime that isn't reusable, and has little to do during its construction, it's usually cheaper to create a new instance whenever one is needed. In that case, you would return `false` from the **CanBePooled** method.

Note Returning `true` from the **CanBePooled** method doesn't guarantee that objects will be recycled; it only gives the MTS run-time environment permission to recycle them. On systems that don't support object pooling, a return value of `true` is ignored. Returning `false` from the **CanBePooled** method guarantees that instances of a component aren't recycled.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl.CanBePooled Method Example

```
import com.ms.mtx.*;

boolean CanBePooled()
{
   // This object should not be recycled,
   // so return false.
   return false;
}
```

IObjectControl.Deactivate Method

Implementing this method allows an MTS object to perform any cleanup required before it's recycled or destroyed. This method is called by the MTS run-time environment whenever an object is deactivated.

Provided By

IObjectContext

void Deactivate ( );.

Remarks

The MTS run-time environment calls the **Deactivate** method whenever an object that supports the **IObjectControl** interface is deactivated. An object is deactivated when it returns from a method in which it called **SetComplete** or **SetAbort**, when the transaction in which it executed is committed or aborted, or when the last client to hold a reference on it releases its reference.

If the component supports recycling (returns `true` from the **CanBePooled** method), you should use the **Deactivate** method to reset the object's state to the state in which the **Activate**method expects to find it. You can also use the **Deactivate** method to release the object's **ObjectContext** reference or to do other context-specific cleanup. Even if an object supports recycling, it can be beneficial to release certain reusable resources in its **Deactivate** method. For example, ODBC provides its own connection pooling. It's more efficient to pool a database connection in a general connection pool where it can be used by other objects than it is to keep it tied to a specific object in an object pool.

Example

See Also

Deactivating Objects, Object Pooling and Recycling

IObjectControl.Deactivate Method Example

```
import com.ms.mtx.*;

void Deactivate( )
{
   // Perform any necessary cleanup here.
}
```

SafeRef Function

Used by an object to obtain a reference to itself that's safe to pass outside its context.

Syntax

Set *safeobject* = **SafeRef**(**Me**)

Part

safeobject
An object variable representing the current object that's safe to pass to another object or client.

Remarks

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call **SafeRef** first and then pass the reference returned by this call.

To use the **SafeRef** function, you must set a reference to Microsoft Transaction Server Type Library (mtxas.dll).

Example

See Also

Passing Object References

SafeRef Function Example

```
Dim anotherObject As New ObjectThatCallsBack
Dim safeMe As My.Class

' Get a safe reference.
Set safeMe = SafeRef(Me)

' Invoke a method on another object, passing the
' safe reference so it can call back.
If Not safeMe Is Nothing Then
   Call anotherObject.CallMeBack(safeMe)
Set safeMe = Nothing
```

GetObjectContext Function

Obtains a reference to the **ObjectContext** that's associated with the current MTS object.

To use the **GetObjectContext** function, you must set a reference to Microsoft Transaction Server Type Library (mtxas.dll).

Syntax

Set *objectcontext* = **GetObjectContext ( )**

Parameters

objectcontext
An object variable that evaluates to the **ObjectContext** belonging to the current object.

Remarks

An object should never attempt to pass its **ObjectContext** reference to another object. If you pass an **ObjectContext** reference to another object, it will no longer be a valid reference.

When an object obtains a reference to its **ObjectContext,** it must release the **ObjectContext** object when it's finished with it.

Example

See Also

Building Scalable Components, Context Objects, **CreateInstance** Method

GetObjectContext Function, CreateInstance Method Example

```
Dim ctxObject As ObjectContext
Dim objAccount As Bank.Account

' Get the object's ObjectContext.
Set ctxObject = GetObjectContext()

' Use it to instantiate another object.
Set objAccount = _
   ctxObject.CreateInstance("Bank.Account")
```

ObjectContext Object

The **ObjectContext** object provides access to the current object's context.

Remarks

To use the **ObjectContext** object, you must set a reference to Microsoft Transaction Server Type Library (mtxas.dll).

You obtain a reference to the **ObjectContext** object by calling the **GetObjectContext** function. As with any COM object, you must release an **ObjectContext** object when you're finished using it, unless it's a local variable.

You can use an object's **ObjectContext** to:

- Declare that the object's work is complete.
- Prevent a transaction from being committed, either temporarily or permanently.
- Instantiate other MTS objects and include their work within the scope of the current object's transaction.
- Find out if a caller is in a particular role.
- Find out if security is enabled.
- Find out if the object is executing within a transaction.
- Retrieve Microsoft Internet Information Server (IIS) built-in objects.

The **ObjectContext** object provides the following methods.

Method	Description
Count	Returns the number of context object properties.
CreateInstance	Instantiates another MTS object.
DisableCommit	Declares that the object hasn't finished its work and that its transactional updates are in an inconsistent state. The object retains its state across method calls, and any attempts to commit the transaction before the object calls **EnableCommit** or **SetComplete** will result in the transaction being aborted.
EnableCommit	Declares that the object's work isn't necessarily finished, but its transactional updates are in a consistent state. This method allows the transaction to be committed, but the object retains its state across method calls until it calls **SetComplete** or **SetAbort**, or until the transaction is completed.
IsCallerInRole	Indicates whether the object's direct caller is in a specified role (either directly or as part of a group).
IsInTransaction	Indicates whether the object is executing within a transaction.
IsSecurityEnabled	Indicates whether security is enabled. MTS security is enabled unless the object is running in the client's process.
Item	Returns a context object property.
Security	Returns a reference to an object's **SecurityProperty** object.
SetAbort	Declares that the object has completed its work and can be deactivated on returning from the currently executing method, but that its transactional updates

	are in an inconsistent state or that an unrecoverable error occurred. This means that the transaction in which the object was executing must be aborted. If any object executing within a transaction returns to its client after calling **SetAbort**, the entire transaction is doomed to abort.
SetComplete	Declares that the object has completed its work and can be deactivated on returning from the currently executing method. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed. When an object that is the root of a transaction calls **SetComplete**, MTS attempts to commit the transaction on return from the current method.

Note When an object calls **DisableCommit**, **EnableCommit**, **SetComplete**, or **SetAbort** from within a method, two flags (Done and Consistent) are set in its **ObjectContext**. (See the following table for an explanation.) These flags aren't evaluated by the MTS run-time environment until the object's currently executing method returns to its caller. This means that an object can call these methods any number of times from within one of its own methods, but the last call before the object returns to its client is the one that will be in effect.

Method	Done	Consistent
SetAbort	TRUE	FALSE
SetComplete	TRUE	TRUE
DisableCommit	FALSE	FALSE
EnableCommit	FALSE	TRUE

The **Done** flag, which allows an object to be deactivated and its transaction to commit or abort, is only evaluated after the object returns from the call that first entered its context. For example, suppose client A calls into object B. Object B calls **SetComplete** and then calls into object C (passing it a safe reference for a callback). Object C calls back to object B, and then object B returns to client A. Object B won't be deactivated when it returns to object C; it will be deactivated when it returns to client A.

See Also

Basic Security Methods, Passing Object References, Context Objects, Transactions, Deactivating Objects

Count Method

Returns the number of context object properties.

Applies To

ObjectContext Object

Syntax

objectcontext.**Count**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Return Value

The number of properties.

Example

CreateInstance Method

Instantiates an MTS object.

Applies To

ObjectContext Object

Syntax

Set *object* = *objectcontext*.**CreateInstance**("*progID*")

Part

object
An object variable that evaluates to an MTS object.

objectcontext
An object variable that represents the **ObjectContext** from which to instantiate the new *object*.

progID
A string expression that is the programmatic ID of the new object's component.

Remarks

CreateInstance creates a COM object. However, the object will have context only if its component is registered with MTS.

When you create an object by using **CreateInstance**, the new object's context is derived from the current object's **ObjectContext** and the declarative properties of the new object's component. The new object always executes within the same activity as the object that created it. If the current object has a transaction, the transaction attribute of the new object's component determines whether or not the new object will execute within the scope of that transaction.

If the component's transaction attribute is set to either **Requires a transaction** or **Supports transactions**, the new object inherits its creator's transaction. If the component's transaction attribute is set to **Requires a new transaction**, MTS initiates a new transaction for the new object. If the component's transaction attribute is set to **Does not support transactions**, the new object doesn't execute under any transaction.

Example

See Also

Creating MTS Objects, Transaction Attributes, MTS Component Requirements

DisableCommit Method

Declares that the object's transactional updates are inconsistent and can't be committed in their present state.

Applies To

ObjectContext Object

Syntax

objectcontext.**DisableCommit**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Remarks

An object that invokes **DisableCommit** is stateful.

You can use the **DisableCommit** method to prevent a transaction from committing prematurely between method calls in a stateful object. When an object invokes **DisableCommit**, it indicates that its work is inconsistent and that it can't complete its work until it receives further method invocations from the client. It also indicates that it needs to maintain its state to perform that work. This prevents the MTS run-time environment from deactivating the object and reclaiming its resources on return from a method call. Once an object has called **DisableCommit**, if a client attempts to commit the transaction before the object has called **EnableCommit** or **SetComplete**, the transaction will abort.

For example, suppose you have a General Ledger component that updates a database. A client makes multiple calls to a General Ledger object to post entries to various accounts. There's an integrity constraint that says the debits must equal the credits when the final method invocation returns, or the transaction must abort. The General Ledger object has an initialization method in which the client informs it of the sequence of calls the client is going to make, and the General Ledger object calls **DisableCommit**. The object maintains its state between calls so that after the final call in the sequence is made the object can make sure the integrity constraint is satisfied before allowing its work to be committed.

Example

See Also

Transactions

DisableCommit Method Example

```
Dim objContext As ObjectContext

Set objContext = GetObjectContext()
objContext.DisableCommit
```

EnableCommit Method

Declares that the current object's work is not necessarily finished, but that its transactional updates are consistent and could be committed in their present form.

Applies To

ObjectContext Object

Syntax

objectcontext.**EnableCommit**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Remarks

When an object calls **EnableCommit**, it allows the transaction in which it's participating to be committed, but it maintains its internal state across calls from its clients until it calls **SetComplete** or **SetAbort** or until the transaction completes.

EnableCommit is the default state when an object is activated. This is why an object should always call **SetComplete** or **SetAbort** before returning from a method, unless you want the object to maintain its internal state for the next call from a client.

Example

See Also

Transactions

EnableCommit Method Example

```
Dim objContext As ObjectContext

Set objContext = GetObjectContext()
objContext.EnableCommit
```

IsCallerInRole Method

Indicates whether an object's direct caller is in a specified role (either individually or as part of a group).

Applies To

ObjectContext Object

Syntax

objectcontext.**IsCallerInRole**(*role*)

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Parameters

objectcontext
An object variable that represents the **ObjectContext** belonging to the current object.

role
A string expression that contains the name of the role in which to determine if the caller is acting.

Return Values

True
Either the caller is in the specified role, or security is not enabled.

False
The caller is not in the specified role.

Remarks

You use this method to determine whether the direct caller of the currently executing method is associated with a specific role. A role is a symbolic name that represents a user or group of users who have specific access permissions to all components in a given package. Developers define roles when they create a component, and roles are mapped to individual users or groups at deployment time.

IsCallerInRole only applies to the direct caller of the currently executing method. (The direct caller is the process calling into the current server process. It can be either a base client process or a server process.) **IsCallerInRole** doesn't apply to the process that initiated the call sequence from which the current method was called, or to any other callers in that sequence.

Because **IsCallerInRole** returns **True** when the object that invokes it is executing in a client's process, it's a good idea to call **IsSecurityEnabled** before calling **IsCallerInRole**. If security isn't enabled, **IsCallerInRole** won't return an accurate result.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

IsCallerInRole, IsSecurityEnabled Methods Example

```
Dim objContext As ObjectContext
Set objContext = GetObjectContext()

If Not objContext Is Nothing Then
   ' Find out if Security is enabled.
   If objContext.IsSecurityEnabled Then
      ' Find out if the caller is in the right role.
      If Not objContext.IsCallerInRole("Manager") Then
         ' If not, do something appropriate here.
      Else
         ' If so, execute the call normally.
      End If
   Else
   ' If security's not enabled, do something
   ' appropriate here.
   End If
End If
```

IsInTransaction Method

Indicates whether the current object is executing in a transaction.

Applies To

ObjectContext Object

Syntax

objectcontext.**IsInTransaction**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Return Values

True
The current object is executing within a transaction.

False
The current object is not executing within a transaction.

Remarks

You can use this method to make sure that an object that requires a transaction never runs without one. For example, if a component that requires a transaction is improperly configured in the MTS Explorer, you can use this method to determine that the object doesn't have a transaction. Then you can return an error to alert the user to the problem, or take whatever action is appropriate.

Example

See Also

Transaction Attributes, Transactions

IsInTransaction Method Example

```
Dim objContext As ObjectContext
Set objContext = GetObjectContext()

If Not objContext Is Nothing Then
   ' Find out if the object is in a transaction.
   If Not objContext.IsInTransaction Then
      ' If not, do something appropriate here.
   End If
End If
```

IsSecurityEnabled Method

Indicates whether or not security is enabled for the current object. MTS security is enabled unless the object is running in the client's process.

Applies To

ObjectContext Object

Syntax

objectcontext.**IsSecurityEnabled**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Return Values

True
Security is enabled for this object.

False
Security is not enabled for this object.

Remarks

MTS security is enabled only if an object is running in a server process. This could be either because the object's component was configured to run in a client's process, or because the component and the client are in the same package. If the object is running in the client's process, there is no security checking and **IsSecurityEnabled** will always return **False**.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

Item Method

Returns a context object property.

Applies To

ObjectContext Object

Syntax

objectcontext.**Item(***name***)**

Part

objectcontext
An object variable that evaluates to the **ObjectContext** associated with the current object.

name
The name of the context object property to be retrieved.

Return Value

The requested context object property.

Remarks

You can use **Item** to retrieve the following Microsoft Internet Information Server (IIS) built-in objects:

- Request
- Response
- Server
- Application
- Session

For more information, see the IIS documentation.

The **Item** method is the default method for a collection. Therefore, the following lines of code are equivalent:

```
oc("Response").Write "<p>" & prop & "</p>"
oc.Item("Response").Write "<p>" & prop & "</p>"
```

Example

Count, Item Methods Example

```
' Get the context object
Dim oc As ObjectContext
Dim str As String
Set oc = GetObjectContext()

' Get the Response object
' Print number of properties
oc("Response").Write "<p>Number of properties: " & oc.Count & "</p>"

' Iterate over properties collection and print the
' names of the properties
For Each prop In oc
    oc("Response").Write "<p>" & prop & "</p>"
Next
```

Security Property

Returns a reference to an object's **SecurityProperty** object.

Applies To

ObjectContext Object

Syntax

Set *objectsecurity* = *objectcontext*.**Security**

Part

objectcontext
An object variable that evaluates to the **ObjectContext** associated with the current object.

objectsecurity
An object variable that evaluates to the **SecurityProperty** object associated with the current object.

Example

See Also

Programmatic Security, Advanced Security Methods

Security Property Example

```
Public Function UsingSecurityMethod() As String

    Dim objCtx As ObjectContext
    Dim objSP As SecurityProperty

    Set objCtx = GetObjectContext()
    Set objSP = objCtx.Security
    UsingSecurityMethod = _
        objSP.GetOriginalCreatorName()

End Function
```

SetAbort Method

Declares that the transaction in which the object is executing must be aborted, and that the object should be deactivated on returning from the currently executing method call.

Applies To

ObjectContext Object

Syntax

objectcontext.**SetAbort**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Remarks

The object is deactivated automatically on return from the method in which it called **SetAbort**. If the object is the root of an automatic transaction, MTS aborts the transaction. If the object is transactional, but not the root of an automatic transaction, the transaction in which it's participating is doomed to abort. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

You can call **SetAbort** in error handlers to ensure that a transaction aborts when an error occurs. You can also call **SetAbort** at the beginning of a method to protect your object from committing prematurely in the event of an unexpected return and then call **SetComplete** just before the method returns, if all goes well.

Example

See Also

Transactions, Context Objects, Deactivating Objects

SetAbort, SetComplete Methods Example

```
Dim ctxObject As ObjectContext
Set ctxObject = GetObjectContext()
On Error GoTo ErrorHandler

' Do some work here. If the work was successful,
' call SetComplete.
ctxObject.SetComplete
Set ctxObject = Nothing
Perform = 0
Exit Function

' If an error occurred, call SetAbort in the error
' handler.
ErrorHandler:
   ctxObject.SetAbort
   Set ctxObject = Nothing
   Perform = -1
   Exit Function
```

SetComplete Method

Declares that the current object has completed its work and should be deactivated when the currently executing method returns to the client. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed.

Applies To

ObjectContext Object

Syntax

objectcontext.**SetComplete**

The *objectcontext* placeholder represents an object variable that evaluates to the **ObjectContext** associated with the current object.

Remarks

The object is deactivated automatically on return from the method in which it called **SetComplete**. If the object is the root of an automatic transaction, MTS attempts to commit the transaction. However, if any object that was participating in the transaction has called **SetAbort**, or has called **DisableCommit** and has not subsequently called **EnableCommit** or **SetComplete**, the transaction will be aborted. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

If an object doesn't need to maintain its state after it returns from a method call, it should call **SetComplete** so that it can be automatically deactivated as soon as it returns and its resources can be reclaimed.

Example

See Also

Transactions, Context Objects, Deactivating Objects

SafeRef Function

Used by an object to obtain a reference to itself that's safe to pass outside its context.

The header file for the **SafeRef** function is mtx.h.

void* SafeRef (
  REFIID *riid*
  UNKNOWN* *pUnk*
);

Parameter

riid
  [in] A reference to the interface ID of the interface that the current object wants to pass to another object or client.

pUnk
  [in] A reference to an interface on the current object.

Return Values

Non-NULL
  A pointer to the interface specified in the *riid* parameter that's safe to pass outside the current object's context.

NULL
  The object is requesting a safe reference on an object other than itself, or the interface requested in the *riid* parameter is not implemented.

Remarks

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call **SafeRef** first and then pass the reference returned by this call. An object should never pass a **this** pointer, or a self-reference obtained through an internal call to **QueryInterface**, to a client or to any other object. Once such a reference is passed outside the object's context, it's no longer a valid reference.

Calling **SafeRef** on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

When a client calls **QueryInterface** on a reference that's safe, MTS automatically ensures that the reference returned to the client is also a safe reference.

An object that obtains a safe reference must release the safe reference when it's finished with it.

Note Safe references have different pointer values than their unsafe counterparts. For example, **this** and the safe version of **this** do not have the same value. It's important to be aware of this when testing whether two pointers refer to the same object. Calling **QueryInterface** for **IID_IUnknown** on each of the pointers and comparing the value of the returned pointers may result in the wrong conclusion. It's possible that both pointers refer to the same object, but that one is a safe reference and the other isn't. If both references are safe references, they can be compared in the usual way. This is only a consideration for MTS objects, because clients should never have access to unsafe references.

Example

See Also

Passing Object References

SafeRef Function Example

```
#include <mtx.h>


IMyInterface* pSafeMyObject = NULL;
IAnotherObject* pOtherObject = NULL;
HRESULT hr;

// Get a safe reference.
pSafeMyObject = SafeRef(IID_IMyInterface,
   (IUnknown*)this);

// Invoke a method on another object, passing the
// safe reference so it can call back.
hr = pOtherObject->CallMeBack(pSafeMyObject);

// Release the safe reference.
safeMyObject->Release();
```

GetObjectContext Function

Obtains a reference to the **IObjectContext** interface on the **ObjectContext** that's associated with the current MTS object.

The header file for the **GetObjectContext** function is mtx.h.

HRESULT GetObjectContext (
  IObjectContext** *ppInstanceContext*
);

Parameters

ppInstanceContext
  [out] A reference to the **IObjectContext** interface on the object's context. If the object's component hasn't been imported into an MTS package, or if **GetObjectContext** is called from a constructor or an **IUnknown** method, this will be set to a NULL pointer.

Return Values

S_OK
  A reference to the **IObjectContext** interface on the current object's context is returned in the *ppInstanceContext* parameter.

E_INVALIDARG
  The argument passed in the *ppInstanceContext* parameter is invalid.

E_UNEXPECTED
  An unexpected error occurred.

CONTEXT_E_NOCONTEXT
  The current object doesn't have a context associated with it, because either the component wasn't imported into a package or the object wasn't created with one of the MTS **CreateInstance** methods. This error will also be returned if the **GetObjectContext** method was called from the constructor or from an **IUnknown** method.

Remarks

An object's context is not accessible from an object's constructor or from any **IUnknown** method (**QueryInterface**, **AddRef**, or **Release**).

An object should never attempt to pass its **ObjectContext** reference to another object. If you pass an **ObjectContext** reference to another object, it will no longer be a valid reference.

When an object obtains a reference to its **ObjectContext,** it must release the **ObjectContext** object when it's finished with it.

Example

See Also

Building Scalable Components, Context Objects, **IObjectContext::CreateInstance** Method

GetObjectContext Function, IObjectContext::CreateInstance Method Example

```
#include <mtx.h>


IObjectContext* pObjectContext = NULL;
IAccount* pAccount = NULL;
HRESULT hr;

// Get the object's ObjectContext.
hr = GetObjectContext(&pObjectContext);

// Use it to instantiate another object.
hr = pObjectContext->CreateInstance(CLSID_CAccount,
   IID_IAccount, (void**)&pAccount);
```

IObjectContext Interface

The **IObjectContext** interface provides access to the current object's context.

Remarks

The header file for the **IObjectContext** interface is mtx.h.

You obtain a reference to the **IObjectContext** interface by calling the **GetObjectContext** function. As with any COM object, you must release an **ObjectContext** object when you're finished using it.

You can use an object's **ObjectContext** to:

- Declare that the object's work is complete.
- Prevent a transaction from being committed, either temporarily or permanently.
- Instantiate other MTS objects and include their work within the scope of the current object's transaction.
- Find out if a caller is in a particular role.
- Find out if security is enabled.
- Find out if the object is executing within a transaction.

The **IObjectContext** interface exposes the following methods.

Method	Description
CreateInstance	Instantiates another MTS object.
DisableCommit	Declares that the object hasn't finished its work and that its transactional updates are in an inconsistent state. The object retains its state across method calls, and any attempts to commit the transaction before the object calls **EnableCommit** or **SetComplete** will result in the transaction being aborted.
EnableCommit	Declares that the object's work isn't necessarily finished, but its transactional updates are in a consistent state. This method allows the transaction to be committed, but the object retains its state across method calls until it calls **SetComplete** or **SetAbort**, or until the transaction is completed.
IsCallerInRole	Indicates whether the object's direct caller is in a specified role (either directly or as part of a group).
IsInTransaction	Indicates whether the object is executing within a transaction.
IsSecurityEnabled	Indicates whether security is enabled. MTS security is enabled unless the object is running in the client's process.
SetAbort	Declares that the object has completed its work and can be deactivated on returning from the currently executing method, but that its transactional updates are in an inconsistent state or that an unrecoverable error occurred. This means that the transaction in which the object was executing must be aborted. If any object executing within a transaction returns to its client after calling **SetAbort**, the entire transaction is doomed to abort.
SetComplete	Declares that the object has completed its work and

can be deactivated on returning from the currently executing method. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed. When an object that is the root of a transaction calls **SetComplete**, MTS attempts to commit the transaction on return from the current method.

Note When an object calls **DisableCommit**, **EnableCommit**, **SetComplete**, or **SetAbort** from within a method, two flags (Done and Consistent) are set in its **ObjectContext**. (See the following table for an explanation.) These flags aren't evaluated by the MTS run-time environment until the object's currently executing method returns to its caller. This means that an object can call these methods any number of times from within one of its own methods, but the last call before the object returns to its client is the one that will be in effect.

Method	Done	Consistent
SetAbort	TRUE	FALSE
SetComplete	TRUE	TRUE
DisableCommit	FALSE	FALSE
EnableCommit	FALSE	TRUE

The **Done** flag, which allows an object to be deactivated and its transaction to commit or abort, is only evaluated after the object returns from the call that first entered its context. For example, suppose client A calls into object B. Object B calls **SetComplete** and then calls into object C (passing it a safe reference for a callback). Object C calls back to object B, and then object B returns to client A. Object B won't be deactivated when it returns to object C; it will be deactivated when it returns to client A.

See Also

Basic Security Methods, Passing Object References, Context Objects, Transactions, Deactivating Objects

IObjectContext::CreateInstance Method

Instantiates an MTS object.

Provided By

IObjectContext

HRESULT IObjectContext::CreateInstance (
    REFCLSID *rclsid*,
    REFIID *riid*,
    LPVOID FAR* *ppvObj*
);

Parameter

rclsid
    [in] A reference to the CLSID of the type of object to instantiate.

riid
    [in] A reference to the interface ID of the interface through which you want to communicate with the new object.

ppvObj
    [out] A reference to the requested interface on the new object.

Return Values

S_OK
    The object was created and a reference to it is returned in the *ppvObj* parameter.

REGDB_E_CLASSNOTREG
    The component specified by *rclsid* is not registered as a COM component.

E_OUTOFMEMORY
    There's not enough memory available to instantiate the object.

E_INVALIDARG
    The argument passed in the *ppvObj* parameter is invalid.

E_UNEXPECTED
    An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object, and the other object calls **CreateInstance** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

Remarks

CreateInstance creates a COM object. However, the object will have context only if its component is registered with MTS.

When you create an object by using **CreateInstance**, the new object's context is derived from the current object's **ObjectContext** and the declarative properties of the new object's component. The new object always executes within the same activity as the object that created it. If the current object has a transaction, the transaction attribute of the new object's component determines whether or not the new object will execute within the scope of that transaction.

If the component's transaction attribute is set to either **Requires a transaction** or **Supports transactions**, the new object inherits its creator's transaction. If the component's transaction attribute is set to **Requires a new transaction**, MTS initiates a new transaction for the new object. If the component's transaction attribute is set to **Does not support transactions**, the new object doesn't execute under any transaction.

If the Microsoft Distributed Transaction Coordinator is not running and the object is transactional, the object is successfully created. However, method calls to that object will fail with

CONTEXT_E_TMNOTAVAILABLE. Objects cannot recover from this condition and should be released.

MTS always uses standard marshaling. Even if a component exposes the **IMarshal** interface, its **IMarshal** methods will never be called by the MTS run-time environment.

You can't create MTS objects as part of an aggregation. In this respect, using **CreateInstance** is like using **CoCreateInstance** and specifying NULL for the controlling **IUnknown** interface (*pUnkOuter*).

Example

See Also

Creating MTS Objects, Transaction Attributes, MTS Component Requirements

IObjectContext::DisableCommit Method

Declares that the object's transactional updates are inconsistent and can't be committed in their present state.

Provided By

IObjectContext

HRESULT IObjectContext::DisableCommit ( );

Return Values

S_OK
: The call to **DisableCommit** succeeded. The object's transactional updates can't be committed until the object calls either **EnableCommit** or **SetComplete**.

E_UNEXPECTED
: An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object and the other object calls **DisableCommit** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

CONTEXT_E_NOCONTEXT
: The current object doesn't have a context associated with it. This is probably because it wasn't created with one of the MTS **CreateInstance** methods.

Remarks

An object that invokes **DisableCommit** is stateful.

You can use the **DisableCommit** method to prevent a transaction from committing prematurely between method calls in a stateful object. When an object invokes **DisableCommit**, it indicates that its work is inconsistent and that it can't complete its work until it receives further method invocations from the client. It also indicates that it needs to maintain its state to perform that work. This prevents the MTS run-time environment from deactivating the object and reclaiming its resources on return from a method call. Once an object has called **DisableCommit**, if a client attempts to commit the transaction before the object has called **EnableCommit** or **SetComplete**, the transaction will abort.

For example, suppose you have a General Ledger component that updates a database. A client makes multiple calls to a General Ledger object to post entries to various accounts. There's an integrity constraint that says the debits must equal the credits when the final method invocation returns, or the transaction must abort. The General Ledger object has an initialization method in which the client informs it of the sequence of calls the client is going to make, and the General Ledger object calls **DisableCommit**. The object maintains its state between calls so that after the final call in the sequence is made the object can make sure the integrity constraint is satisfied before allowing its work to be committed.

Example

See Also

Transactions

IObjectContext::DisableCommit Method Example

```
#include <mtx.h>


IObjectContext* pObjectContext = NULL;
HRESULT hr;

hr = GetObjectContext(&pObjectContext);
hr = pObjectContext->DisableCommit();
```

IObjectContext::EnableCommit Method

Declares that the current object's work is not necessarily finished, but that its transactional updates are consistent and could be committed in their present form.

Provided By

IObjectContext

HRESULT IObjectContext::EnableCommit ( );

Return Values

S_OK
: The call to **EnableCommit** succeeded and the object's transactional updates can now be committed.

E_UNEXPECTED
: An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object and the other object calls **EnableCommit** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

Remarks

When an object calls **EnableCommit**, it allows the transaction in which it's participating to be committed, but it maintains its internal state across calls from its clients until it calls **SetComplete** or **SetAbort** or until the transaction completes.

EnableCommit is the default state when an object is activated. This is why an object should always call **SetComplete** or **SetAbort** before returning from a method, unless you want the object to maintain its internal state for the next call from a client.

Example

See Also

Transactions

IObjectContext::EnableCommit Method Example

```
#include <mtx.h>

IObjectContext* pObjectContext = NULL;
HRESULT hr;

hr = GetObjectContext(&pObjectContext);
hr = pObjectContext->EnableCommit();
```

IObjectContext::IsCallerInRole Method

Indicates whether an object's direct caller is in a specified role (either individually or as part of a group).

Provided By

IObjectContext

```
HRESULT IObjectContext::IsCallerInRole (
    BSTR bstrRole,
    BOOL* pfIsInRole
);
```

Parameters

bstrRole
[in] The name of the role in which you want to determine whether the caller is acting.

pfIsInRole
[out] TRUE if the caller is in the specified role, FALSE if not. This parameter will also be set to TRUE if security is not enabled.

Return Values

S_OK
The role specified in the *bstrRole* parameter is a recognized role, and the Boolean result returned in the *pfIsInRole* parameter indicates whether or not the caller is in that role.

CONTEXT_E_ROLENOTFOUND
The role specified in the *bstrRole* parameter does not exist.

E_INVALIDARG
One or more of the arguments passed in is invalid.

E_UNEXPECTED
An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object and the other object calls **IsCallerInRole** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

Remarks

You use this method to determine whether the direct caller of the currently executing method is associated with a specific role. A role is a symbolic name that represents a user or group of users who have specific access permissions to all components in a given package. Developers define roles when they create a component, and roles are mapped to individual users or groups at deployment time.

IsCallerInRole only applies to the direct caller of the currently executing method. (The direct caller is the process calling into the current server process. It can be either a base client process or a server process.) **IsCallerInRole** doesn't apply to the process that initiated the call sequence from which the current method was called, or to any other callers in that sequence.

Because **IsCallerInRole** returns TRUE when the object that invokes it is executing in a client's process, it's a good idea to call **IsSecurityEnabled** before calling **IsCallerInRole**. If security isn't enabled, **IsCallerInRole** won't return an accurate result.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

IObjectContext::IsCallerInRole, IObjectContext::IsSecurityEnabled Methods Example

```
#include <mtx.h>

IObjectContext* pObjectContext = NULL;
BSTR stRole = SysAllocString(L"Manager");
VARIANT_BOOL fIsInRole;
HRESULT hr;

hr = GetObjectContext(&pObjectContext);

// Find out if security is enabled.
if (pObjectContext->IsSecurityEnabled()) {
   //Then find out if the caller is in the right role.
   fIsInRole = pObjectContext->IsCallerInRole
      (stRole, &fIsInRole)
   SysFreeString(stRole);
   if (!fIsInRole) {
      // If not, do something appropriate here.
   }
}
else {
   // If security's not enabled, do something
   // appropriate here.
}
```

IObjectContext::IsInTransaction Method

Indicates whether the current object is executing in a transaction.

Provided By

IObjectContext

BOOL IObjectContext::IsInTransaction ( );

Return Values

TRUE
 The current object is executing within a transaction.

FALSE
 The current object is not executing within a transaction.

Remarks

You can use this method to make sure that an object that requires a transaction never runs without one. For example, if a component that requires a transaction is improperly configured in the MTS Explorer, you can use this method to determine that the object doesn't have a transaction. Then you can return an error to alert the user to the problem, or take whatever action is appropriate.

Example

See Also

Transaction Attributes, Transactions

IObjectContext::IsInTransaction Method Example

```
#include <mtx.h>

IObjectContext* pObjectContext = NULL;
VARIANT_BOOL fInTransaction;
HRESULT hr;

hr = GetObjectContext(&pObjectContext);

// Find out if the object is in a transaction.
fInTransaction = pObjectContext->IsInTransaction();

if (!fInTransaction) {
   // If not, do something appropriate here.
}
```

IObjectContext::IsSecurityEnabled Method

Indicates whether or not security is enabled for the current object. MTS security is enabled unless the object is running in the client's process.

Provided By

IObjectContext

BOOL IObjectContext::IsSecurityEnabled ( );

Return Values

TRUE
  Security is enabled for this object.

FALSE
  Security is not enabled for this object.

Remarks

MTS security is enabled only if an object is running in a server process. This could be either because the object's component was configured to run in a client's process, or because the component and the client are in the same package. If the object is running in the client's process, there is no security checking and **IsSecurityEnabled** will always return FALSE.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

IObjectContext::SetAbort Method

Declares that the transaction in which the object is executing must be aborted, and that the object should be deactivated on returning from the currently executing method call.

Provided By

IObjectContext

HRESULT IObjectContext::SetAbort ( );

Return Values

S_OK
: The call to **SetAbort** succeeded and the transaction will be aborted.

E_UNEXPECTED
: An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object and the other object calls **SetAbort** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

Remarks

The object is deactivated automatically on return from the method in which it called **SetAbort**. If the object is the root of an automatic transaction, MTS aborts the transaction. If the object is transactional, but not the root of an automatic transaction, the transaction in which it's participating is doomed to abort. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

You can call **SetAbort** in error handlers to ensure that a transaction aborts when an error occurs. You can also call **SetAbort** at the beginning of a method to protect your object from committing prematurely in the event of an unexpected return and then call **SetComplete** just before the method returns, if all goes well.

Example

See Also

Transactions, Context Objects, Deactivating Objects

IObjectContext::SetAbort, IObjectContext::SetComplete Methods Example

```
#include <mtx.h>


IObjectContext* pObjectContext = NULL;
HRESULT hr;

hr = GetObjectContext(&pObjectContext);
// Do some work here.
// If the work was successful, call SetComplete.
if (SUCCEEDED(hr)) {
   if (pObjectContext)
      pObjectContext->SetComplete();
}
// Otherwise, call SetAbort.
else {
   if (pObjectContext)
      pObjectContext->SetAbort();
}
```

IObjectContext::SetComplete Method

Declares that the current object has completed its work and should be deactivated when the currently executing method returns to the client. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed.

Provided By

IObjectContext

HRESULT IObjectContext::SetComplete ( );

Return Values

S_OK
    The call to **SetComplete** succeeded.

E_UNEXPECTED
    An unexpected error occurred. This can happen if one object passes its **IObjectContext** pointer to another object and the other object calls **SetComplete** using this pointer. An **IObjectContext** pointer is not valid outside the context of the object that originally obtained it.

Remarks

The object is deactivated automatically on return from the method in which it called **SetComplete**. If the object is the root of an automatic transaction, MTS attempts to commit the transaction. However, if any object that was participating in the transaction has called **SetAbort**, or has called **DisableCommit** and has not subsequently called **EnableCommit** or **SetComplete**, the transaction will be aborted. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

If an object doesn't need to maintain its state after it returns from a method call, it should call **SetComplete** so that it can be automatically deactivated as soon as it returns and its resources can be reclaimed.

Example

See Also

Transactions, Context Objects, Deactivating Objects

MTx.SafeRef Method

Used by an object to obtain a reference to itself that's safe to pass outside its context.

SafeRef is a static method of the **MTx** class, which is declared in the package `com.ms.mtx.`

Note The class **MTx** has only static methods and has no public constructor. You can't create an instance of this class.

IUnknown SafeRef (
  IUnknown *obj*
);

Parameter

obj
  [in] A reference to an interface on the current object.

Return Value

A reference to the **IUnknown** interface on the current object that's safe to pass outside the current object's context.

Remarks

When an MTS object wants to pass a self-reference to a client or another object (for example, for use as a callback), it should always call **SafeRef** first and then pass the reference returned by this call. An object should never pass a reference to **this** to a client or to any other object. Once such a reference is passed outside the object's context, it's no longer a valid reference.

Regardless of the interface ID you pass to **SafeRef**, it always returns the **IUnknown** interface on the object that calls it. You should immediately cast the returned value to the interface that you want to pass outside the object.

Calling **SafeRef** on a reference that is already safe returns the safe reference unchanged, except that the reference count on the interface is incremented.

Example

See Also

Passing Object References

MTx.SafeRef Method Example

```
import com.ms.mtx.*;


IMyInterface safeMyObject = null;
IAnotherObject someOtherObject = null;

// Get a safe reference.
safeMyObject = (IMyInterface) MTx.SafeRef(this);

// Invoke a method on another object, passing the
// safe reference so it can call back.
someOtherObject.CallMeBack(safeMyObject);
```

MTx.GetObjectContext Method

Obtains a reference to the **IObjectContext** interface on the **ObjectContext** that's associated with the current MTS object.

GetObjectContext is a static method of the **MTx** class, which is declared in the package `com.ms.mtx.`

Note The **MTx** class has only static methods and has no public constructor. You can't create an instance of this class.

IObjectContext GetObjectContext ( );

Return Value

A reference to the **IObjectContext** interface on the current object's context. **GetObjectContext** will return null if it is called from a constructor or finalizer, or if the object's component hasn't been imported into an MTS package.

Remarks

An object should never attempt to pass its **ObjectContext** reference to another object. If you pass an **ObjectContext** reference to another object, it will no longer be a valid reference.

Example

See Also

Building Scalable Components, Context Objects, **IObjectContext.CreateInstance** Method

MTx.GetObjectContext Method, IObjectContext.CreateInstance Method Example

```
import com.ms.mtx.*;


IAccount account = null;

// Get the object's ObjectContext and
// use it to instantiate another object.
account = (IAccount) MTx.GetObjectContext().
   CreateInstance(CAccount.clsid, IAccount.iid);
```

IObjectContext Interface

The **IObjectContext** interface provides access to the current object's context.

Remarks

The **IObjectContext** interface is declared in the package `com.ms.mtx.`

You obtain a reference to the **IObjectContext** interface by calling the **MTx.GetObjectContext** method. As with any COM object, you must release an **ObjectContext** object when you're finished using it, unless it's a local variable.

You can use an object's **ObjectContext** to:

- Declare that the object's work is complete.
- Prevent a transaction from being committed, either temporarily or permanently.
- Instantiate other MTS objects and include their work within the scope of the current object's transaction.
- Find out if a caller is in a particular role.
- Find out if security is enabled.
- Find out if the object is executing within a transaction.

The **IObjectContext** interface exposes the following methods.

Method	Description
CreateInstance	Instantiates another MTS object.
DisableCommit	Declares that the object hasn't finished its work and that its transactional updates are in an inconsistent state. The object retains its state across method calls, and any attempts to commit the transaction before the object calls **EnableCommit** or **SetComplete** will result in the transaction being aborted.
EnableCommit	Declares that the object's work isn't necessarily finished, but its transactional updates are in a consistent state. This method allows the transaction to be committed, but the object retains its state across method calls until it calls **SetComplete** or **SetAbort**, or until the transaction is completed.
IsCallerInRole	Indicates whether the object's direct caller is in a specified role (either directly or as part of a group).
IsInTransaction	Indicates whether the object is executing within a transaction.
IsSecurityEnabled	Indicates whether security is enabled. MTS security is enabled unless the object is running in the client's process.
SetAbort	Declares that the object has completed its work and can be deactivated on returning from the currently executing method, but that its transactional updates are in an inconsistent state or that an unrecoverable error occurred. This means that the transaction in which the object was executing must be aborted. If any object executing within a transaction returns to its client after calling **SetAbort**, the entire transaction is doomed to abort.

SetComplete	Declares that the object has completed its work and can be deactivated on returning from the currently executing method. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed. When an object that is the root of a transaction calls **SetComplete**, MTS attempts to commit the transaction on return from the current method.

Note When an object calls **DisableCommit**, **EnableCommit**, **SetComplete**, or **SetAbort** from within a method, two flags (Done and Consistent) are set in its **ObjectContext**. (See the following table for an explanation.) These flags aren't evaluated by the MTS run-time environment until the object's currently executing method returns to its caller. This means that an object can call these methods any number of times from within one of its own methods, but the last call before the object returns to its client is the one that will be in effect.

Method	Done	Consistent
SetAbort	TRUE	FALSE
SetComplete	TRUE	TRUE
DisableCommit	FALSE	FALSE
EnableCommit	FALSE	TRUE

The **Done** flag, which allows an object to be deactivated and its transaction to commit or abort, is only evaluated after the object returns from the call that first entered its context. For example, suppose client A calls into object B. Object B calls **SetComplete** and then calls into object C (passing it a safe reference for a callback). Object C calls back to object B, and then object B returns to client A. Object B won't be deactivated when it returns to object C; it will be deactivated when it returns to client A.

See Also

Basic Security Methods, Passing Object References, Context Objects, Transactions, Deactivating Objects

IObjectContext.CreateInstance Method

Instantiates an MTS object.

Provided By

IObjectContext Interface

IUnknown CreateInstance (
    _Guid *clsid*,
    _Guid *iid*,
);

Parameter

clsid
    [in] The clsid of the type of object to instantiate.
iid
    [in] Any interface that's implemented by the object you want to instantiate.

Return Value

A reference to the **IUnknown** interface on the newly created object.

Remarks

CreateInstance creates a COM object. However, the object will have context only if its component is registered with MTS.

When you create an object by using **CreateInstance**, the new object's context is derived from the current object's **ObjectContext** and the declarative properties of the new object's component. The new object always executes within the same activity as the object that created it. If the current object has a transaction, the transaction attribute of the new object's component determines whether or not the new object will execute within the scope of that transaction.

If the component's transaction attribute is set to either **Requires a transaction** or **Supports transactions**, the new object inherits its creator's transaction. If the component's transaction attribute is set to **Requires a new transaction**, MTS initiates a new transaction for the new object. If the component's transaction attribute is set to **Does not support transactions**, the new object doesn't execute under any transaction.

CreateInstance always returns the **IUnknown** interface on the newly instantiated object. You should immediately cast the returned value to the interface through which you want to communicate with the new object. The interface ID you pass in the *iid* parameter doesn't have to be the same interface as the one to which you cast the returned value, but it must be an interface that's implemented by the object you're instantiating.

MTS always uses standard marshaling. Even if a component exposes the **IMarshal** interface, its **IMarshal** methods will never be called by the MTS run-time environment.

You can't create MTS objects as part of an aggregation.

Example

See Also

Creating MTS Objects, Transaction Attributes, MTS Component Requirements

IObjectContext.DisableCommit Method

Declares that the object's transactional updates are inconsistent and can't be committed in their present state.

Provided By

IObjectContext

void DisableCommit ( );

Remarks

An object that invokes **DisableCommit** is stateful.

You can use the **DisableCommit** method to prevent a transaction from committing prematurely between method calls in a stateful object. When an object invokes **DisableCommit**, it indicates that its work is inconsistent and that it can't complete its work until it receives further method invocations from the client. It also indicates that it needs to maintain its state to perform that work. This prevents the MTS run-time environment from deactivating the object and reclaiming its resources on return from a method call. Once an object has called **DisableCommit**, if a client attempts to commit the transaction before the object has called **EnableCommit** or **SetComplete**, the transaction will abort.

For example, suppose you have a General Ledger component that updates a database. A client makes multiple calls to a General Ledger object to post entries to various accounts. There's an integrity constraint that says the debits must equal the credits when the final method invocation returns, or the transaction must abort. The General Ledger object has an initialization method in which the client informs it of the sequence of calls the client is going to make, and the General Ledger object calls **DisableCommit**. The object maintains its state between calls so that after the final call in the sequence is made the object can make sure the integrity constraint is satisfied before allowing its work to be committed.

Example

See Also

Transactions

IObjectContext.DisableCommit Method Example

```
import com.ms.mtx.*;

MTx.GetObjectContext().DisableCommit();
```

IObjectContext.EnableCommit Method

Declares that the current object's work is not necessarily finished, but that its transactional updates are consistent and could be committed in their present form.

Provided By

IObjectContext

void EnableCommit ( );

Remarks

When an object calls **EnableCommit**, it allows the transaction in which it's participating to be committed, but it maintains its internal state across calls from its clients until it calls **SetComplete** or **SetAbort** or until the transaction completes.

EnableCommit is the default state when an object is activated. This is why an object should always call **SetComplete** or **SetAbort** before returning from a method, unless you want the object to maintain its internal state for the next call from a client.

Example

See Also

Transactions

IObjectContext.EnableCommit Method Example

```
import com.ms.mtx.*;

MTx.GetObjectContext().EnableCommit();
```

IObjectContext.IsCallerInRole Method

Indicates whether an object's direct caller is in a specified role (either individually or as part of a group).

Provided By

IObjectContext

boolean IsCallerInRole (
String *bstrRole*
);

Parameters

bstrRole
[in] The name of the role in which you want to determine whether the caller is acting.

Return Values

`true`
Either the caller is in the specified role, or security is not enabled.

`false`
The caller is not in the specified role.

Remarks

You use this method to determine whether the direct caller of the currently executing method is associated with a specific role. A role is a symbolic name that represents a user or group of users who have specific access permissions to all components in a given package. Developers define roles when they create a component, and roles are mapped to individual users or groups at deployment time.

IsCallerInRole only applies to the direct caller of the currently executing method. (The direct caller is the process calling into the current server process. It can be either a base client process or a server process.) **IsCallerInRole** doesn't apply to the process that initiated the call sequence from which the current method was called, or to any other callers in that sequence.

Because **IsCallerInRole** returns `true` when the object that invokes it is executing in a client's process, it's a good idea to call **IsSecurityEnabled** before calling **IsCallerInRole**. If security isn't enabled, **IsCallerInRole** won't return an accurate result.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

IObjectContext.IsCallerInRole, IObjectContext.IsSecurityEnabled Methods Example

```
import com.ms.mtx.*;

IObjectContext objContext = null;

objContext = MTx.GetObjectContext();

// Find out if Security is enabled.
if (objContext.IsSecurityEnabled()) {
   //Then find out if the caller is in the right role.
   if (!objContext.IsCallerInRole("Manager")) {
      // If not, do something appropriate here.
   }
}
else {
   // If security's not enabled, do something
   // appropriate here.
}
```

IObjectContext.IsInTransaction Method

Indicates whether the current object is executing in a transaction.

Provided By

IObjectContext

boolean IsInTransaction ( );

Return Values

`true`
  The current object is executing within a transaction.

`false`
  The current object is not executing within a transaction.

Remarks

You can use this method to make sure that an object that requires a transaction never runs without one. For example, if a component that requires a transaction is improperly configured in the MTS Explorer, you can use this method to determine that the object doesn't have a transaction. Then you can return an error to alert the user to the problem, or take whatever action is appropriate.

Example

See Also

Transaction Attributes, Transactions

IObjectContext.IsInTransaction Method Example

```
import com.ms.mtx.*;


// Find out if the object is in a transaction.
if (!MTx.GetObjectContext().IsInTransaction()) {
   // If not, do something appropriate here.
}
```

IObjectContext.IsSecurityEnabled Method

Indicates whether or not security is enabled for the current object. MTS security is enabled unless the object is running in the client's process.

Provided By

IObjectContext

boolean IsSecurityEnabled ( );

Return Values

`true`
  Security is enabled for this object.
`false`
  Security is not enabled for this object.

Remarks

MTS security is enabled only if an object is running in a server process. This could be either because the object's component was configured to run in a client's process, or because the component and the client are in the same package. If the object is running in the client's process, there is no security checking and **IsSecurityEnabled** will always return `false`.

Example

See Also

Programmatic Security, Basic Security Methods, Secured Components

IObjectContext.SetAbort Method

Declares that the transaction in which the object is executing must be aborted, and that the object should be deactivated on returning from the currently executing method call.

Provided By

IObjectContext

void SetAbort ( );

Remarks

The object is deactivated automatically on return from the method in which it called **SetAbort**. If the object is the root of an automatic transaction, MTS aborts the transaction. If the object is transactional, but not the root of an automatic transaction, the transaction in which it's participating is doomed to abort. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

You can call **SetAbort** in error handlers to ensure that a transaction aborts when an error occurs. You can also call **SetAbort** at the beginning of a method to protect your object from committing prematurely in the event of an unexpected return and then call **SetComplete** just before the method returns, if all goes well.

Example

See Also

Transactions, Context Objects, Deactivating Objects

IObjectContext.SetAbort, IObjectContext.SetComplete Methods Example

```
import com.ms.mtx.*;


boolean success = false;
// Do some work here.
// If the work was successful, call SetComplete
if (success)
   MTx.GetObjectContext().SetComplete();
// Otherwise, call SetAbort.
else
   MTx.GetObjectContext().SetAbort();
```

IObjectContext.SetComplete Method

Declares that the current object has completed its work and should be deactivated when the currently executing method returns to the client. For objects that are executing within the scope of a transaction, it also indicates that the object's transactional updates can be committed.

Provided By

IObjectContext

void SetComplete ( );

Remarks

The object is deactivated automatically on return from the method in which it called **SetComplete**. If the object is the root of an automatic transaction, MTS attempts to commit the transaction. However, if any object that was participating in the transaction has called **SetAbort**, or has called **DisableCommit** and has not subsequently called **EnableCommit** or **SetComplete**, the transaction will be aborted. (An object is the root of a transaction if the MTS run-time environment has to initiate a new transaction for it. This is the case when the component that provides the object is configured to require a transaction and the object's creator doesn't have one, or when the component is configured to require a new transaction.)

If an object doesn't need to maintain its state after it returns from a method call, it should call **SetComplete** so that it can be automatically deactivated as soon as it returns and its resources can be reclaimed.

Example

See Also

Transactions, Context Objects, Deactivating Objects

IObjectContextActivity Interface

The **IObjectContextActivity** interface is used to retrieve a unique identifier associated with the current activity. This activity identifier is a GUID, and is only valid for the lifetime of the current activity.

Remarks

The header file for the **IObjectContextActivity** is mtx.h. You must also link mtxguid.lib to your project to use this interface.

You obtain a reference to an object's **IObjectContextActivity** interface by calling **QueryInterface** on the object's **ObjectContext**. For example:

```
m_pIObjectContext->QueryInterface
    (IID_IObjectContextActivity,
    (void**)&m_pIObjectContextActivity));
```

The **IObjectContextActivity** interface provides the following methods.

Method	Description
GetActivityId	Retrieves the GUID associated with the current activity.

IObjectContextActivity::GetActivityId Method

Retrieves the GUID associated with the current activity.

Provided By

IObjectContextActivity

HRESULT IObjectContextActivity::GetActivityId(
GUID * *pActivityId***);**

Parameters

pActivityId
[out] A reference to the GUID associated with the current activity.

Return Values

S_OK
The GUID of the current activity is returned in the parameter *pActivityId*.

E_INVALIDARG
The argument passed in the *pActivityId* parameter is a NULL pointer.

E_UNEXPECTED
An unexpected error occurred.

Example

GetActivityId Method Example

```
#include <mtx.h>

HRESULT hr = S_OK;
IObjectContext *pIObjectContext = NULL;
IObjectContextActivity *pIObjectContextActivity = NULL;
GUID activityId;

// Get object context
hr = GetObjectContext(&pIObjectContext);
// Get IObjectContextActivity interface
hr = pIObjectContext->
    QueryInterface(IID_IObjectContextActivity,
    (void**)&pIObjectContextActivity);
// Use IObjectContextActivity to retrieve
// the activity GUID.
hr = pIObjectContextActivity->
    GetActivityId(&activityId);

// Do something with the activity GUID here.

// Release the IObjectContextActivity
// and the IObjectContext pointers
pIObjectContextActivity->Release();
pIObjectContext->Release();
```

SharedPropertyGroupManager Object

The **SharedPropertyGroupManager** object is used to create shared property groups and to obtain access to existing shared property groups.

Remarks

To use the **SharedPropertyGroupManager** object, you must set a reference to the Shared Property Manager Type Library (mtxspm.dll).

You can access the **SharedPropertyGroupManager** object by using either the **CreateObject** function or the **CreateInstance** method of the **ObjectContext** object. It makes no difference which you use.

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. You can't use global variables in a distributed environment because of concurrency and name collision issues. The Shared Property Manager eliminates name collisions by providing shared property groups, which establish unique name spaces for the shared properties they contain. The Shared Property Manager also implements locks and semaphores to protect shared properties from simultaneous access, which could result in lost updates and could leave the properties in an unpredictable state.

Shared properties can be shared only by objects running in the same process. If you want instances of different components to share properties, you have to install the components in the same MTS package. Because there is a risk that administrators will move components from one package to another, it's safest to limit the use of a shared property group to instances of components that are defined in the same DLL.

It's also important for components sharing properties to have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same package.

You should always instantiate the **SharedPropertyGroupManager**, **SharedPropertyGroup**, and **SharedProperty** objects from MTS objects rather than from a base client. If a base client creates shared property groups and properties, the shared properties are inside the base client's process, not in a server process. This means MTS objects can't share the properties unless the objects, too, are running in the client's process (which is generally not a good idea).

Note When you set the isolation mode to **LockMethod**, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a non-MTS object doesn't have an **ObjectContext**.

The **SharedPropertyGroupManager** object provides the following methods and properties.

Method	Description
CreatePropertyGroup	Creates a new **SharedPropertyGroup** with a string name as an identifier. If a group with the specified name already exists, **CreatePropertyGroup** returns a reference to the existing group.

Group	Returns a reference to an existing shared property group, given a string name by which it can be identified.

See Also

Sharing State

CreatePropertyGroup Method

Creates and returns a reference to a new shared property group. If a property group with the specified name already exists, **CreatePropertyGroup** returns a reference to the existing group.

Applies To

SharedPropertyGroupManager Object

Syntax

Set *propertygroup* = *sharedpropertygroupmanager*.**CreatePropertyGroup**(*name*, *dwIsoMode*, *dwrelmode, fExists*)

Parameters

propertygroup
An object variable that evaluates to a **SharedPropertyGroup** object.

sharedpropertygroupmanager
An object variable that represents the **SharedPropertyGroupManager** with which to create the shared property group.

name
A string expression that contains the name of the shared property group to create.

dwIsoMode
A **Long** value that specifies the isolation mode for the properties in the new shared property group. See the table that lists *dwIsoMode* constants later in this topic. If the value of the *fExists* parameter is set to **True** on return from this method, the *dwIsoMode* value you passed in is ignored and the value returned in this parameter is the isolation mode that was assigned when the property group was created.

dwRelMode
A **Long** value that specifies the release mode for the properties in the new shared property group. See the table that lists *dwRelMode* constants later in this topic. If the value of the *fExists* parameter is set to **True** on return from this method, the *dwRelMode* value you passed in is ignored and the value returned in this parameter is the release mode that was assigned when the property group was created.

fExists
A **Boolean** value that's set to **True** on return from this method if the shared property group specified in the *name* parameter existed prior to this call, and **False** if the property group was created by this call.

Settings

The following constants are used in the *dwIsoMode* parameter to specify the effective isolation mode for a shared property group.

Constant	Value	Description
LockSetGet	0	Default. Locks a property during a **Value** call, assuring that every get or set operation on a shared property is atomic. This ensures that two clients can't read or write to the same property at the same time, but it doesn't prevent other clients from concurrently accessing other properties in the same group.
LockMethod	1	Locks all of the properties in the shared property group for exclusive use by the caller as long as

the caller's current method is executing.

This is the appropriate mode to use when there are interdependencies among properties, or in cases where a client may have to update a property immediately after reading it before it can be accessed again.

Note When you set the isolation mode to **LockMethod**, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a base client doesn't have an **ObjectContext**.

The following constants are used in the *dwRelMode* parameter to specify the effective release mode for a shared property group.

Constant	Value	Description
Standard	0	When all clients have released their references on the property group, the property group is automatically destroyed.
Process	1	The property group isn't destroyed until the process in which it was created has terminated. You must still release all **SharedPropertyGroup** objects by setting them to **Nothing**.

Remarks

The **CreatePropertyGroup** method sets the value in *fExists* to **True** if the property group it returns existed prior to the current call. This occurs when another object in the same process has already called **CreatePropertyGroup** with the same property group name. The **CreatePropertyGroup** method sets the value in *fExists* to **False** if the returned property group was created by the current call.

The isolation mode and release mode are assigned when the property group is originally created and aren't changed if a subsequent call passes different values in these parameters. The caller should always check the value of *fExists* on return from this method. If *fExists* is set to **True**, the caller should check the values returned in *dwIsoMode* and *dwRelMode* to determine the isolation and release modes in effect for the property group. For example:

```
Dim isolationMode As Long
Dim releaseMode As Long

Set isolationMode = LockMethod
Set releaseMode = Process
Set spmGroup = spmMgr.CreatePropertyGroup _
   ("Counter", isolationMode, releaseMode, fExists)

If fExists Then
   If isolationMode <> LockMethod _
      Or releaseMode <> Process Then
      ' Do something appropriate.
   EndIf
EndIf
```

You can pass the constants, **LockGetSet** or **LockMethod** as the *dwIsoMode* argument, and **Standard** or **Process** as the *dwRelMode* argument, directly to the **CreatePropertyGroup** method. However, when you pass a constant instead of a variable, the **CreatePropertyGroup** method can't return the isolation and release modes currently in effect if the requested property group already

exists.

Note An object should never attempt to pass a shared property group reference to another object. If the reference is passed outside of the object that acquired it, it's no longer a valid reference.

Example

See Also

Sharing State, **IObjectContext** Interface, **SharedPropertyGroup** Object

Group Property

Returns a reference to an existing shared property group.

Applies To

ISharedPropertyGroupManager Interface

Syntax

Set *propertygroup* = *sharedpropertygroupmanager*.**Group**(*name*)

Parameters

propertygroup
An object variable that evaluates to a **SharedPropertyGroup** object.

sharedpropertygroupmanager
An object variable that represents the **SharedPropertyGroupManager** for the current process.

name
A string expression that contains the name of the shared property group to retrieve.

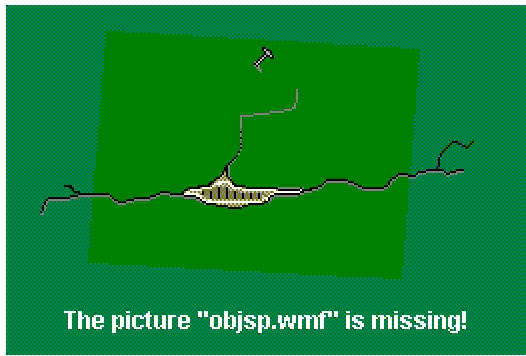
Example

See Also

Sharing State, **SharedPropertyGroupManager** Object

SharedPropertyGroup Object

The **SharedPropertyGroup** object is used to create and access the shared properties in a shared property group.



Remarks

To use the **SharedPropertyGroup** object, you must set a reference to the Shared Property Manager Type Library (mtxspm.dll)

You can create a **SharedPropertyGroup** object with the **CreatePropertyGroup** method of the **SharedPropertyGroupManager** object.

As with any COM object, you must release a **SharedPropertyGroup** object when you're finished using it, unless it's a local variable. For example:

```
Set myPropertyGroup = Nothing
```

The **SharedPropertyGroup** object provides the following methods and properties.

Method/Property	Description
CreateProperty	Creates a new shared property identified by a string expression that's unique within its property group.
CreatePropertyByPosition	Creates a new shared property identified by a numeric index within its property group.
Property	Returns a reference to a shared property, given the string name by which the property is identified.
PropertyByPosition	Returns a reference to a shared property, given its numeric index in the shared property group.

See Also

Sharing State, **ISharedPropertyGroupManager** Object

CreateProperty Method

Creates and returns a reference to a new **SharedProperty** with a specified name. If a shared property by that name already exists, **CreateProperty** returns a reference to the existing property.

Applies To

SharedPropertyGroup Object

Syntax

Set *property* = *propertygroup*.**CreateProperty**(*name, fExists*)

Parameters

propertygroup
: An object variable that represents the **SharedPropertyGroup** to which the new **SharedProperty** object will belong.

property
: An object variable that evaluates to a **SharedProperty** object.

name
: A string expression that contains the name of the property to create. You can use this name later to obtain a reference to this property.

fExists
: A Boolean value that's set to **True** on return from this method if the shared property specified in the *name* parameter existed prior to this call, and **False** if the property was created by this call.

Remarks

When you create a shared property, its value is set to the default, which is 0.

If you create a shared property with the **CreateProperty** method, you can access that property only by using **Property**. You can't assign a numeric index to the same property and then access it by using **PropertyByPosition**.

The same shared property group can contain some shared property objects that are identified by name and others that are identified by position.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

CreatePropertyByPosition Method

Creates a new shared property identified by a numeric index that's unique within the property group. If a shared property with the specified index already exists, **CreatePropertyByPosition** returns a reference to the existing one.

Applies To

SharedPropertyGroup Object

Syntax

Set *property* = *propertygroup*.**CreatePropertyByPosition** (*index, fExists*)

Parameters

property
An object variable that evaluates to a **SharedProperty** object.

propertygroup
An object variable that represents the **SharedPropertyGroup** to which the new **SharedProperty** object will belong.

index
A **Long** value that represents the numeric index within the **SharedPropertyGroup** by which the new property will be referenced. You can use this index later to retrieve the shared property with **PropertyByPosition**.

fExists
A **Boolean** value. If *fExists* is set to **True** on return from this method, the shared property specified by *index* existed prior to this call. If it's set to **False**, the property was created by this call.

Remarks

When you create a shared property, its value is set to the default, which is 0.

If you create a **SharedProperty** object with the **CreatePropertyByPosition** method, you can access that property only by using **PropertyByPosition**. You can't assign a string name to the same property and then access it by using **Property**. Accessing a property by position is faster than accessing a property by using a string name because it requires less overhead.

The same shared property group can contain some **SharedProperty** objects that are identified by position and others that are identified by name.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

CreatePropertyByPosition Method Example

```
Dim spmMgr As SharedPropertyGroupManager
Dim spmGroup As SharedPropertyGroup
Dim spmPropNextNumber As SharedProperty
Dim bExists As Boolean
Dim iNextValue As Integer

' Create the SharedPropertyGroupManager,
' SharedPropertyGroup, and SharedProperty.
Set spmMgr = CreateObject _
   ("MTxSpm.SharedPropertyGroupManager.1")
Set spmGroup = spmMgr.CreatePropertyGroup _
   ("Counter", LockSetGet, Process, bExists)
Set spmPropNextNumber = _
   spmGroup.CreatePropertyByPosition(0, bExists)

' Get the next number and increment it.
iNextValue = spmPropNextNumber.Value
spmPropNextNumber.Value = _
   spmPropNextNumber.Value + 1
```

Property Property

Returns a reference to an existing shared property identified by a string name.

Applies To

SharedPropertyGroup Object

Syntax

Set *property* = *propertygroup*.**Property**(*name)*

Parameters

propertygroup
: An object variable that represents the **SharedPropertyGroup** to which the **SharedProperty** object belongs.

property
: An object variable that evaluates to a **SharedProperty** object.

name
: A string expression that contains the name of the shared property to retrieve.

Remarks

You can use only **Property** to access properties that were created with the **CreateProperty** method. To access properties that were created with the **CreatePropertyByPosition** method, use **PropertyByPosition**.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method

Property, Group Properties Example

```
Dim spmMgr As SharedPropertyGroupManager
Dim spmGroup As SharedPropertyGroup
Dim spmPropNextNumber As SharedProperty
Dim iNextValue As Integer

' Get the SharedPropertyGroupManager,
' SharedPropertyGroup, and SharedProperty.
Set spmMgr = CreateObject _
   ("MTxSpm.SharedPropertyGroupManager.1")
Set spmGroup = spmMgr.Group("Counter")
Set spmPropNextNumber = spmGroup.Property("Next")

' Get the next number and increment it.
iNextValue = spmPropNextNumber.Value
spmPropNextNumber.Value = _
   spmPropNextNumber.Value + 1
```

PropertyByPosition Property

Returns a reference to an existing shared property identified by its numeric index within the property group.

Applies To

SharedPropertyGroup Object

Syntax

Set *sharedproperty* = *propertygroup*.**PropertyByPosition**(*index*)

Parameters

propertygroup
An object variable that represents the **SharedPropertyGroup** object to which the **SharedProperty** object belongs.

sharedproperty
An object variable that evaluates to a **SharedProperty** object.

index
A **Long** value that represents the numeric index within the **SharedPropertyGroup** of the property to retrieve.

Remarks

You can use only **PropertyByPosition** to access properties that were created with the **CreatePropertyByPosition** method. To access properties that were created with the **CreateProperty** method, use **Property**.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

PropertyByPosition Property Example

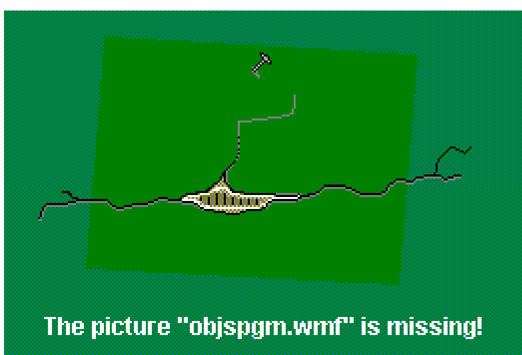
```
Dim spmMgr As SharedPropertyGroupManager
Dim spmGroup As SharedPropertyGroup
Dim spmPropNextNumber As SharedProperty
Dim iNextValue As Integer

' Get the SharedPropertyGroupManager,
' SharedPropertyGroup, and SharedProperty.
Set spmMgr = CreateObject _
   ("MTxSpm.SharedPropertyGroupManager.1")
Set spmGroup = spmMgr.Group("Counter")
Set spmPropNextNumber = spmGroup.PropertyByPosition(0)

' Get the next number and increment it.
iNextValue = spmPropNextNumber.Value
spmPropNextNumber.Value = _
   spmPropNextNumber.Value + 1
```

SharedProperty Object

The **SharedProperty** object is used to set or retrieve the value of a shared property. A shared property can contain any data type that can be represented by a variant.



Remarks

To use the **SharedProperty** object, you must set a reference to the Shared Property Manager Type Library (mtxspm.dll).

You can create a **SharedProperty** object with the **CreateProperty** method or the **CreatePropertyByPosition** method.

A **SharedProperty** object can be created or accessed only from within a **SharedPropertyGroup**.

As with any COM object, you must release a **SharedProperty** object when you're finished using it, unless it's a local variable. For example:

```
Set myProperty = Nothing
```

The **SharedProperty** object provides the following property.

Property	Description
Value	Sets or retrieves the value of a shared property.

See Also

Sharing State, MTS Supported Variant Types

Value Property

Sets or retrieves the value of a shared property.

Applies To

SharedProperty Object

Syntax

property.**Value** = *value*

Parameters

property
An object variable that represents a **SharedProperty** object.

value
A **Variant** containing the value to assign to the **SharedProperty** object, or the **SharedProperty**'s current value.

Example

See Also

MTS Supported Variant Types, Sharing State

CreatePropertyGroup Method, CreateProperty Method, Value Property Example

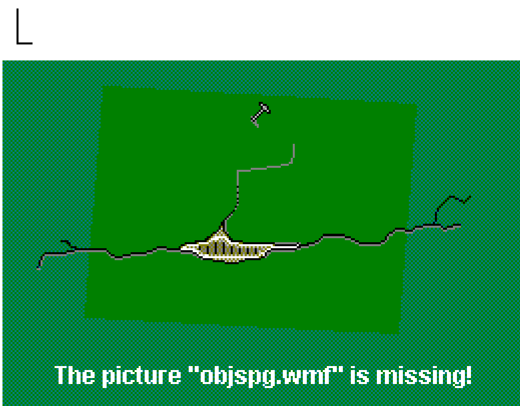
```
Dim spmMgr As SharedPropertyGroupManager
Dim spmGroup As SharedPropertyGroup
Dim spmPropNextNumber As SharedProperty
Dim bExists As Boolean
Dim iNextValue As Integer

' Create the SharedPropertyGroupManager,
' SharedPropertyGroup, and SharedProperty.
Set spmMgr = CreateObject _
    ("MTxSpm.SharedPropertyGroupManager.1")
Set spmGroup = spmMgr.CreatePropertyGroup _
    ("Counter", LockSetGet, Process, bExists)
Set spmPropNextNumber = _
   spmGroup.CreateProperty("Next", bExists)

' Get the next number and increment it.
iNextValue = spmPropNextNumber.Value
spmPropNextNumber.Value = _
   spmPropNextNumber.Value + 1
```

ISharedPropertyGroupManager Interface

The **ISharedPropertyGroupManager** interface is used to create shared property groups and to obtain access to existing shared property groups.



Remarks

The header file for the **ISharedPropertyGroupManager** interface is mtxspm.h. You must also link mtxguid.lib to your project to use this interface.

You can access the **ISharedPropertyGroupManager** interface by creating an instance of the **SharedPropertyGroupManager** by using either **IObjectContext::CreateInstance** or **CoCreateInstance**. It makes no difference which you use.

CreateInstance method of the **ObjectContext** object. It makes no difference which you use.

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. You can't use global variables in a distributed environment because of concurrency and name collision issues. The Shared Property Manager eliminates name collisions by providing shared property groups, which establish unique name spaces for the shared properties they contain. The Shared Property Manager also implements locks and semaphores to protect shared properties from simultaneous access, which could result in lost updates and could leave the properties in an unpredictable state.

Shared properties can be shared only by objects running in the same process. If you want instances of different components to share properties, you have to install the components in the same MTS package. Because there is a risk that administrators will move components from one package to another, it's safest to limit the use of a shared property group to instances of components that are defined in the same DLL.

It's also important for components sharing properties to have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same package.

You should always instantiate the **SharedPropertyGroupManager**, **SharedPropertyGroup**, and **SharedProperty** objects from MTS objects rather than from a base client. If a base client creates shared property groups and properties, the shared properties are inside the base client's process, not in a server process. This means MTS objects can't share the properties unless the objects, too, are running in the client's process (which is generally not a good idea).

Note When you set the isolation mode to **LockMethod**, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a non-MTS object doesn't have an **ObjectContext**.

The **ISharedPropertyGroupManager** interface exposes the following methods and properties.

Method	Description
CreatePropertyGroup	Creates a new **SharedPropertyGroup** with a string name as an identifier. If a group with the specified name already

	exists, **CreatePropertyGroup** returns a reference to the existing group.
get_Group	Returns a reference to an existing shared property group, given a string name by which it can be identified.
get__NewEnum	Returns a reference to an enumerator that iterates through a list of all the shared property groups in a given process.

See Also

Sharing State

ISharedPropertyGroupManager::CreatePropertyGroup Method

Creates and returns a reference to a new shared property group. If a property group with the specified name already exists, **CreatePropertyGroup** returns a reference to the existing group.

Provided By

ISharedPropertyGroupManager Interface

HRESULT ISharedPropertyGroup::CreatePropertyGroup (
  BSTR *name*,
  LONG* *plIsoMode*,
  LONG* *plRelMode*,
  VARIANT_BOOL* *pfExists,*
  ISharedPropertyGroup** *ppGroup*,
);

Parameters

name
  [in] The name of the shared property group to create.

plIsoMode
  [in, out] A reference to a LONG that specifies the isolation mode for the properties in the new shared property group. See the table that lists *plIsoMode* constants later in this topic. If the value of the *pfExists* parameter is set to VARIANT_TRUE on return from this method, the *plIsoMode* value you passed in is ignored and the value returned in this parameter is the isolation mode that was assigned when the property group was created.

plRelMode
  [in, out] A reference to a LONG that specifies the release mode for the properties in the new shared property group. See the table that lists *plRelMode* constants later in this topic. If the value of the *pfExists* parameter is set to VARIANT_TRUE on return from this method, the *plRelMode* value you passed in is ignored and the value returned in this parameter is the release mode that was assigned when the property group was created.

pfExists
  [out] A reference to a BOOL that's set to VARIANT_TRUE on return from this method if the shared property group specified in the *name* parameter existed prior to this call, and VARIANT_FALSE if the property group was created by this call.

ppGroup
  [out] A reference to a shared property group identified by the BSTR passed in the *name* parameter*,* or NULL if an error is encountered.

Settings

The following constants are used in the *plIsoMode* parameter to specify the effective isolation mode for a shared property group.

Constant	Value	Description
LockSetGet	0	Default. Locks a property during a **get_Value** or **put_Value**call, assuring that every get or set operation on a shared property is atomic. This ensures that two clients can't read or write to the same property at the same time, but it doesn't prevent other clients from concurrently accessing other properties in the same group.
LockMethod	1	Locks all of the properties in the shared property

group for exclusive use by the caller as long as the caller's current method is executing.

This is the appropriate mode to use when there are interdependencies among properties, or in cases where a client may have to update a property immediately after reading it before it can be accessed again.

Note When you set the isolation mode to **LockMethod**, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a base client doesn't have an **ObjectContext**.

The following constants are used in the *plRelMode* parameter to specify the effective release mode for a shared property group.

Constant	Value	Description
Standard	0	When all clients have released their references on the property group, the property group is automatically destroyed. (This is the default COM mode.)
Process	1	The property group isn't destroyed until the process in which it was created has terminated. (Objects that hold references on a property group must still call **Release** on their references).

Return Values

S_OK
: A reference to the shared property group specified in the *name* parameter is returned in the *ppGroup* parameter.

CONTEXT_E_NOCONTEXT
: The caller isn't executing under the MTS run-time environment. A caller must be executing under MTS to use the Shared Property Manager.

E_INVALIDARG
: At least one of the parameters is invalid, or the same object is attempting to create the same property group more than once.

Remarks

The **CreatePropertyGroup** method sets the value in *pfExists* to VARIANT_TRUE if the property group it returns in the *ppGroup* parameter existed prior to the current call. This occurs when another object in the same process has already called **CreatePropertyGroup** with the same property group name. The **CreatePropertyGroup** method sets the value in *pfExists* to VARIANT_FALSE if the returned property group was created by the current call.

The isolation mode and release mode are assigned when the property group is originally created and aren't changed if a subsequent call passes different values in these parameters. The caller should always check the value of *pfExists* on return from this method. If *pfExists* is set to VARIANT_TRUE, the caller should check the values returned in *plIsoMode* and *plRelMode* to determine the isolation and release modes in effect for the property group. For example:

```
hr = pPropGpMgr->CreatePropertyGroup(stName,
   &lIsolationMode, &lReleaseMode, &fAlreadyExists,
   &pPropGp);
if (fAlreadyExists) {
```

```
    if ((lIsolationMode != LockMethod) ||
        (lReleaseMode != Process)) {
        // Do something appropriate.
    }
}
If*
```

Note An object should never attempt to pass a shared property group reference to another object. If the reference is passed outside of the object that acquired it, it's no longer a valid reference.

Example

See Also

Sharing State, **IObjectContext** Interface, **ISharedPropertyGroup** Interface

ISharedPropertyGroupManager::get_Group Method

Returns a reference to an existing shared property group.

Provided By

ISharedPropertyGroupManager Interface

HRESULT ISharedPropertyGroupManager::get_Group (
  BSTR *name*,
  ISharedPropertyGroup** *ppGroup*,
);

Parameters

name
  [in] The name of the shared property group to retrieve.

ppGroup
  [out] A reference to the shared property group specified in the *name* parameter, or NULL if the property group doesn't exist.

Return Values

S_OK
  The shared property group exists, and a reference to it is returned in the *ppGroup* parameter.

E_INVALIDARG
  The shared property group with the name specified in the *name* parameter doesn't exist.

Example

See Also

Sharing State, **ISharedPropertyGroupManager** Interface

ISharedPropertyGroupManager::get__NewEnum Method

Returns a reference to an enumerator that you can use to iterate through all the shared property groups in a process.

Provided By

ISharedPropertyGroupManager Interface

HRESULT ISharedPropertyGroupManager::get__NewEnum (
IUnknown** *ppEnumerator*
);

Parameters

ppEnumerator
[out] A reference to the **IUnknown** interface on a new enumerator object that you can use to iterate through the list of all the shared property groups in the process.

Return Values

S_OK
A reference to the requested enumerator is returned in the *ppEnumerator* parameter.

Remarks

You use the **get__NewEnum** method to obtain a reference to an enumerator object. You should immediately call **QueryInterface** on the returned **IUnknown** for the **IEnumVARIANT** interface. This interface exposes several methods you can use to iterate through a list of BSTRs representing shared property group names. Once you have a name, you can use the **get_Group** method to obtain a reference to the shared property group it represents.

As with any COM object, you must release an enumerator object when you're finished using it. When you enumerate the shared property groups, all groups will be included. However, if you then call **CreatePropertyGroup** to add a new group, the existing enumerator won't include the new group even if you call **Reset** or **Clone**. To include the new group, you must create a new enumerator by calling **get__NewEnum** again.

Note **get__NewEnum** has two underscore characters between **get** and **NewEnum**.

Example

See Also

ISharedPropertyGroupManager::get_Group Method, **ISharedPropertyGroup Interface**

ISharedPropertyGroupManager::get__NewEnum Method Example

```
#include <mtxspm.h>

ISharedPropertyGroupManager* pspgm = NULL;
IUnknown* pUnknown = NULL;
IEnumVARIANT* pEnum = NULL;
VARIANT v;
ULONG cElementsFetched;
int i;
HRESULT hr;

// Get the enumerator object.
hr = pspgm->get__NewEnum(&pUnknown);

// Query for the IEnumVARIANT interface.
hr = pUnknown->QueryInterface(IID_IEnumVARIANT, (void**) &pEnum);

// Use the enumerator to iterate through
// the property group names.
for(i = 0; i < 10; i++)
{
   VariantInit(&v);
   pEnum->Next(1, &v, &cElementsFetched);
   // Do something with the returned
   // property group names.
}
```

ISharedPropertyGroup Interface

The **ISharedPropertyGroup** interface is used to create and access the shared properties in a shared property group.

Remarks

The header file for the **ISharedPropertyGroup** interface is mtxspm.h. You must also link mtxguid.lib to your project to use this interface.

You can access the **ISharedPropertyGroup** interface by creating a **SharedPropertyGroup** object with the **ISharedPropertyGroupManager::CreatePropertyGroup** method.

As with any COM object, you must release a **SharedPropertyGroup** object when you're finished using it.

The **ISharedPropertyGroup** interface exposes the following methods.

CreateProperty	Creates a new shared property identified by a string expression that's unique within its property group.
CreatePropertyByPosition	Creates a new shared property identified by a numeric index within its property group.
get_Property	Returns a reference to a shared property, given the string name by which the property is identified.
get_PropertyByPosition	Returns a reference to a shared property, given its numeric index in the shared property group.

See Also

Sharing State, **ISharedPropertyGroupManager** Interface

ISharedPropertyGroup::CreateProperty Method

Creates and returns a reference to a new **SharedProperty** with a specified name. If a shared property by that name already exists, **CreateProperty** returns a reference to the existing property.

Provided By

ISharedPropertyGroup Interface

HRESULT ISharedPropertyGroup::CreateProperty (
  BSTR *name*,
  VARIANT_BOOL* *pfExists;*
  ISharedProperty** *ppProp*,
);

Parameters

name
  [in] The name of the property to create. You can use this name later to obtain a reference to this property by using the **get_Property** method.

pfExists
  [out] A reference to a Boolean value that's set to VARIANT_TRUE on return from this method if the shared property specified in the *name* parameter existed prior to this call, and VARIANT_FALSE if the property was created by this call.

ppProp
  [out] A reference to a **SharedProperty** object with the name specified in the *name* parameter, or NULL if an error is encountered.

Return Values

S_OK
  A reference to a shared property with the name specified in the *name* parameter is returned in the parameter *ppProp*.

E_INVALIDARG
  One or more of the arguments passed in is invalid.

Remarks

When you create a shared property, its value is set to the default, which is a VARIANT of type VT_I4, with a value of 0.

If you create a shared property with the **CreateProperty** method, you can access that property only by using the **get_Property** method. You can't assign a numeric index to the same property and then access it by using the **get_PropertyByPosition** method.

The same shared property group can contain some shared property objects that are identified by name and others that are identified by position.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup::CreatePropertyByPosition Method

Creates a new shared property identified by a numeric index that's unique within the property group. If a shared property with the specified index already exists, **CreatePropertyByPosition** returns a reference to the existing one.

Provided By

ISharedPropertyGroup Interface

HRESULT ISharedPropertyGroup::CreatePropertyByPosition (
  INT *index*,
  VARIANT_BOOL* *pfExists;*
  ISharedProperty** *ppProp*,
);

Parameters

index
  [in] The numeric index within the **SharedPropertyGroup** by which the new property will be referenced. You can use this index later to retrieve the shared property with the **get_PropertyByPosition** method.

pfExists
  [out] A reference to a Boolean value. If *pfExists* is set to VARIANT_TRUE on return from this method, the shared property specified by *index* existed prior to this call. If it's set to VARIANT_FALSE, the property was created by this call.

ppProp
  [out] A reference to a shared property object identified by the numeric index passed in the *index* parameter, or NULL if an error is encountered.

Return Values

S_OK
  A reference to the shared property occupying the position specified in the *index* parameter is returned in the *ppProp* parameter.

E_INVALIDARG
  One or more of the arguments passed in is invalid.

Remarks

When you create a shared property, its value is set to the default, which is a VARIANT of type VT_I4, with a value of 0.

If you create a **SharedProperty** object with the **CreatePropertyByPosition** method, you can access that property only by using the **get_PropertyByPosition** method. You can't assign a string name to the same property and then access it by using the **get_Property** method. Accessing a property by position is faster than accessing a property by using a string name because it requires less overhead.

The same shared property group can contain some **SharedProperty** objects that are identified by position and others that are identified by name.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup::CreatePropertyByPosition Method Example

```
#include <mtx.h>
#include <mtxspm.h>


IObjectContext* pObjectContext = NULL;
ISharedPropertyGroupManager* pPropGpMgr = NULL;
ISharedPropertyGroup* pPropGp = NULL;
ISharedProperty* pPropNextNum = NULL;
VARIANT_BOOL fAlreadyExists = VARIANT_FALSE;
LONG lIsolationMode = LockMethod;
LONG lReleaseMode = Process;
BSTR stName;
VARIANT vNext;
LONG lNextValue = 0L;
HRESULT hr = S_OK;

hr = GetObjectContext(&pObjectContext);

// Create the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
hr = pObjectContext->CreateInstance
   (CLSID_SharedPropertyGroupManager,
   IID_ISharedPropertyGroupManager,
   (void**)&pPropGpMgr);

stName = SysAllocString(L"Counter");
hr = pPropGpMgr->CreatePropertyGroup(stName,
   &lIsolationMode, &lReleaseMode, &fAlreadyExists,
   &pPropGp);
SysFreeString(stName);

hr = pPropGp->CreatePropertyByPosition
   (0, &fAlreadyExists, &pPropNextNum);

// Get the next number and increment the counter.
VariantInit(&vNext);
vNext.vt = VT_I4;
hr = pPropNextNum->get_Value(&vNext);
lNextValue = vNext.lVal++;
hr = pPropNextNum->put_Value(vNext);
```

ISharedPropertyGroup::get_Property Method

Returns a reference to an existing shared property identified by a string name.

Provided By

ISharedPropertyGroup Interface

HRESULT ISharedPropertyGroup::get_Property (
  BSTR *name*,
  ISharedProperty** *ppProperty*,
);

Parameters

name
  [in] The name of the shared property to retrieve.

ppProperty
  [out] A reference to the shared property specified in the *name* parameter, or NULL if the property doesn't exist.

Return Values

S_OK
  The shared property specified by *name* was found and a reference to it is returned in the *ppProperty* parameter.

E_INVALIDARG
  Either the argument passed in the *ppProperty* parameter was a null pointer, or there is no property in the shared property group with the name specified in the *name* parameter.

Remarks

You can use only the **get_Property** method to access properties that were created with the **CreateProperty** method. To access properties that were created with the **CreatePropertyByPosition** method, use the **get_PropertyByPosition** method.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method

ISharedPropertyGroupManager::get_Group, ISharedPropertyGroup::get_Property Methods Example

```
#include <mtx.h>
#include <mtxspm.h>


IObjectContext* pObjectContext = NULL;
ISharedPropertyGroupManager* pPropGpMgr = NULL;
ISharedPropertyGroup* pPropGp = NULL;
ISharedProperty* pPropNextNum = NULL;
BSTR stName, stNextNumber;
VARIANT vNext;
LONG lNextValue = 0L;
HRESULT hr = S_OK;

hr = GetObjectContext(&pObjectContext);

// Get the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
hr = pObjectContext->CreateInstance
    (CLSID_SharedPropertyGroupManager,
    IID_ISharedPropertyGroupManager,
    (void**)&pPropGpMgr);

stName = SysAllocString(L"Counter");
hr = pPropGpMgr->get_Group(stName, &pPropGp);
SysFreeString(stName);

stNextNumber = SysAllocString(L"NextNum");
hr = pPropGp->get_Property
    (stNextNumber, &pPropNextNum);
SysFreeString(stNextNumber);

// Get the next number and increment the counter.
VariantInit(&vNext);
vNext.vt = VT_I4;
hr = pPropNextNum->get_Value(&vNext);
lNextValue = vNext.lVal++;
hr = pPropNextNum->put_Value(vNext);
```

ISharedPropertyGroup::get_PropertyByPosition Method

Returns a reference to an existing shared property identified by its numeric index within the property group.

Provided By

ISharedPropertyGroup Interface

HRESULT ISharedPropertyGroup::get_PropertyByPosition (
  INT *index*,
  ISharedProperty** *ppProperty*,
);

Parameters

index
  [in] The numeric index within the **SharedPropertyGroup** of the property to retrieve.

ppProperty
  [out] A reference to the shared property specified by the *index* parameter, or NULL if the property doesn't exist.

Return Values

S_OK
  The shared property specified by *index* was found and a reference to it is returned in the *ppProperty* parameter.

E_INVALIDARG
  Either the argument passed in the *ppProperty* parameter was a null pointer, or there is no property in the shared property group with the index number specified in the *index* parameter.

Remarks

You can use only the **get_PropertyByPosition** method to access properties that were created with the **CreatePropertyByPosition** method. To access properties that were created with the **CreateProperty** method, use the **get_Property** method.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup::get_PropertyByPosition Method Example

```
#include <mtx.h>
#include <mtxspm.h>


IObjectContext* pObjectContext = NULL;
ISharedPropertyGroupManager* pPropGpMgr = NULL;
ISharedPropertyGroup* pPropGp = NULL;
ISharedProperty* pPropNextNum = NULL;
BSTR stName;
VARIANT vNext;
LONG lNextValue = 0L;
HRESULT hr = S_OK;

hr = GetObjectContext(&pObjectContext);

// Get the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
hr = pObjectContext->CreateInstance
   (CLSID_SharedPropertyGroupManager,
   IID_ISharedPropertyGroupManager,
   (void**)&pPropGpMgr);

stName = SysAllocString(L"Counter");
hr = pPropGpMgr->get_Group(stName, &pPropGp);
SysFreeString(stName);

hr = pPropGp->get_PropertyByPosition
   (0, &pPropNextNum);

// Get the next number and increment the counter.
VariantInit(&vNext);
vNext.vt = VT_I4;
hr = pPropNextNum->get_Value(&vNext);
lNextValue = vNext.lVal++;
hr = pPropNextNum->put_Value(vNext);
```

ISharedProperty Interface

The **ISharedProperty** interface is used to set or retrieve the value of a shared property. A shared property can contain any data type that can be represented by a variant.

Remarks

The header file for the **ISharedProperty** interface is mtxspm.h. You must also link mtxguid.lib to your project to use this interface.

You can access the **ISharedProperty** interface by creating a **SharedProperty** object with the **ISharedPropertyGroup::CreateProperty** method or the **ISharedPropertyGroup::CreatePropertyByPosition** method.

A **SharedProperty** object can be created or accessed only from within a **SharedPropertyGroup**.

As with any COM object, you must release a **SharedProperty** object when you're finished using it.

The **ISharedProperty** interface exposes the following methods.

Method	Description
get_Value	Retrieves the value of a shared property.
put_Value	Sets the value of a shared property.

See Also

Sharing State, MTS Supported Variant Types

ISharedProperty::get_Value Method

Retrieves the value of a shared property.

Provided By

ISharedProperty Interface

HRESULT ISharedProperty::get_Value (
  VARIANT* *pVal*
);

Parameters

pVal
  [out] A reference to a variant in which the value of the shared property will be returned.

Return Values

S_OK
  A reference to a VARIANT containing the value of the shared property is returned in the *pVal* parameter.

E_INVALIDARG
  The argument passed in the *pval* parameter is invalid.

Example

See Also

MTS Supported Variant Types, Sharing State

ISharedProperty::put_Value Method

Assigns a value to a shared property.

Provided By

ISharedProperty Interface

HRESULT ISharedProperty::put_Value (
  VARIANT *value*
);

Parameters

value
  [in] A VARIANT containing the value to assign to the **SharedProperty** object.

Return Values

S_OK
  The shared property's value has been set to *value*.

E_INVALIDARG
  The argument passed in the *value* parameter has the VT_BYREF bit set.

DISP_E_ARRAYISLOCKED
  The argument passed in the *value* parameter contains an array that's locked.

DISP_E_BADVARTYPE
  The argument passed in the *value* parameter isn't a valid VARIANT type.

Example

See Also

MTS Supported Variant Types, Sharing State

CreatePropertyGroup, CreateProperty, put_Value, get_Value Methods Example

```
#include <mtx.h>
#include <mtxspm.h>


IObjectContext* pObjectContext = NULL;
ISharedPropertyGroupManager* pPropGpMgr = NULL;
ISharedPropertyGroup* pPropGp = NULL;
ISharedProperty* pPropNextNum = NULL;
VARIANT_BOOL fAlreadyExists = VARIANT_FALSE;
LONG lIsolationMode = LockMethod;
LONG lReleaseMode = Process;
LONG lNextValue = 0L;
BSTR stName, stNextNumber;
VARIANT vNext;
HRESULT hr = S_OK;

hr = GetObjectContext(&pObjectContext);

// Create the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
hr = pObjectContext->CreateInstance
   (CLSID_SharedPropertyGroupManager,
   IID_ISharedPropertyGroupManager,
   (void**)&pPropGpMgr);

stName = SysAllocString(L"Counter");
hr = pPropGpMgr->CreatePropertyGroup(stName,
   &lIsolationMode, &lReleaseMode, &fAlreadyExists,
   &pPropGp);
SysFreeString(stName);

stNextNumber = SysAllocString(L"NextNum");
hr = pPropGp->CreateProperty
   (stNextNumber, &fAlreadyExists, &pPropNextNum);
SysFreeString(stNextNumber);

// Get the next number and increment the counter.
VariantInit(&vNext);
vNext.vt = VT_I4;
hr = pPropNextNum->get_Value(&vNext);
lNextValue = vNext.lVal++;
hr = pPropNextNum->put_Value(vNext);
```

ISharedPropertyGroupManager Interface

The **ISharedPropertyGroupManager** interface is used to create shared property groups and to obtain access to existing shared property groups.

Remarks

The **ISharedPropertyGroupManager** interface is declared in the package `com.ms.mtx.`

You can access the **ISharedPropertyGroupManager** interface by creating an instance of the **SharedPropertyGroupManager** by using either **IObjectContext.CreateInstance** or **new SharedPropertyGroupManager**. It makes no difference which you use.

CreateInstance method of the **ObjectContext** object. It makes no difference which you use.

The Shared Property Manager is a resource dispenser that you can use to share state among multiple objects within a server process. You can't use global variables in a distributed environment because of concurrency and name collision issues. The Shared Property Manager eliminates name collisions by providing shared property groups, which establish unique name spaces for the shared properties they contain. The Shared Property Manager also implements locks and semaphores to protect shared properties from simultaneous access, which could result in lost updates and could leave the properties in an unpredictable state.

Shared properties can be shared only by objects running in the same process. If you want instances of different components to share properties, you have to install the components in the same MTS package. Because there is a risk that administrators will move components from one package to another, it's safest to limit the use of a shared property group to instances of components that are defined in the same DLL.

It's also important for components sharing properties to have the same activation attribute. If two components in the same package have different activation attributes, they generally won't be able to share properties. For example, if one component is configured to run in a client's process and the other is configured to run in a server process, their objects will usually run in different processes, even though they're in the same package.

You should always instantiate the **SharedPropertyGroupManager**, **SharedPropertyGroup**, and **SharedProperty** objects from MTS objects rather than from a base client. If a base client creates shared property groups and properties, the shared properties are inside the base client's process, not in a server process. This means MTS objects can't share the properties unless the objects, too, are running in the client's process (which is generally not a good idea).

Note When you set the isolation mode to LOCKMODE_METHOD, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a non-MTS object doesn't have an **ObjectContext**.

The **ISharedPropertyGroupManager** interface exposes the following methods and properties.

Method	Description

CreatePropertyGroup	Creates a new **SharedPropertyGroup** with a string name as an identifier. If a group with the specified name already exists, **CreatePropertyGroup** returns a reference to the existing group.
getGroup	Returns a reference to an existing shared property group, given a string name by which it can be identified.
get_NewEnum	Returns a reference to an enumerator that iterates through a list of all the shared property groups in a given process.

See Also

Sharing State

ISharedPropertyGroupManager.CreatePropertyGroup Method

Creates and returns a reference to a new shared property group. If a property group with the specified name already exists, **CreatePropertyGroup** returns a reference to the existing group.

Provided By

ISharedPropertyGroupManager Interface

ISharedPropertyGroup CreatePropertyGroup (
String *name*,
int[] *lockmode*,
int[] *releasemode*,
boolean[] *exists,*
);

Parameters

name
[in] The name of the shared property group to create.

lockmode
[in, out] An array of one integer that specifies the isolation mode for the properties in the new shared property group. See the table that lists *lockmode* constants later in this topic. If the value of the *exists* parameter is set to `true` on return from this method, the *lockmode* value you passed in is ignored and the value returned in this parameter is the isolation mode that was assigned when the property group was created.

releasemode
[in, out] An array of one integer that specifies the release mode for the properties in the new shared property group. See the table that lists *releasemode* constants later in this topic. If the value of the *exists* parameter is set to `true` on return from this method, the *releasemode* value you passed in is ignored and the value returned in this parameter is the release mode that was assigned when the property group was created.

exists
[out] An array of one boolean that's set to `true` on return from this method if the shared property group specified in the *name* parameter existed prior to this call, and `false` if the property group was created by this call.

Settings

The following constants are used in the *lockmode* parameter to specify the effective isolation mode for a shared property group. These constants are static final members of the **ISharedPropertyGroupManager** interface.

Constant	Value	Description
LOCKMODE _SETGET	0	Default. Locks a property during a **getValue** or **putValue**call, assuring that every get or set operation on a shared property is atomic. This ensures that two clients can't read or write to the same property at the same time, but it doesn't prevent other clients from concurrently accessing other properties in the same group.
LOCKMODE _METHOD	1	Locks all of the properties in the shared property group for exclusive use by the caller as long as the caller's current method is executing. This is the appropriate mode to use when there

are interdependencies among properties, or in cases where a client may have to update a property immediately after reading it before it can be accessed again.

Note When you set the isolation mode to LOCKMODE_METHOD, the Shared Property Manager requires access to the calling object's **ObjectContext**. You can't use this isolation mode to create a shared property group from within an object's constructor or from a non-MTS object because **ObjectContext** isn't available during object construction and a base client doesn't have an **ObjectContext**.

The following constants are used in the *releasemode* parameter to specify the effective release mode for a shared property group. These constants are static final members of the **ISharedPropertyGroupManager** interface.

Constant	Value	Description
RELEASE MODE_ STANDARD	0	When all clients have released their references on the property group, the property group is automatically destroyed. (This is the default COM mode.)
RELEASE MODE_ PROCESS	1	The property group isn't destroyed until the process in which it was created has terminated.

Return Value

A reference to a shared property group identified by the string expression passed in the *name* parameter, or `null` if an error is encountered.

Remarks

The **CreatePropertyGroup** method sets the value in e*xists* to `true` if the property group it returns existed prior to the current call. This occurs when another object in the same process has already called **CreatePropertyGroup** with the same property group name. The **CreatePropertyGroup** method sets the value in e*xists* to `false` if the returned property group was created by the current call.

The isolation mode and release mode are assigned when the property group is originally created and aren't changed if a subsequent call passes different values in these parameters. The caller should always check the value of e*xists* on return from this method. If *exists* is set to `true`, the caller should check the values returned in *lockmode* and *releasemode* to determine the isolation and release modes in effect for the property group. For example:

```
propGp = propGpMgr.CreatePropertyGroup
   ("Counter", aiIsolationMode,
   aiReleaseMode, afAlreadyExists);
if (afAlreadyExists[0]) {
   if ((aiIsolationMode[0] !=
      ISharedPropertyGroupManager.LOCKMODE_METHOD) ||
      (aiReleaseMode[0] ISharedPropertyGroupManager.
      RELEASEMODE_PROCESS)) {
      // Do something appropriate.
   }
}
If*
```

Note An object should never attempt to pass a shared property group reference to another object. If the reference is passed outside of the object that acquired it, it's no longer a valid reference.

Example

See Also

Sharing State, **IObjectContext** Interface, **ISharedPropertyGroup** Interface

ISharedPropertyGroupManager.getGroup Method

Returns a reference to an existing shared property group.

Provided By

ISharedPropertyGroupManager Interface

ISharedPropertyGroup getGroup (
    String *name*,
);

Parameters

name
    [in] The name of the shared property group to retrieve.

Return Value

A reference to the shared property group specified in the *name* parameter, or `null` if the property group doesn't exist.

Example

See Also

Sharing State, **ISharedPropertyGroupManager** Interface

ISharedPropertyGroupManager.get_NewEnum Method

Returns a reference to an enumerator that you can use to iterate through all the shared property groups in a process.

Provided By

ISharedPropertyGroupManager Interface

IUnknown get_NewEnum ( );

Return Value

A reference to the **IUnknown** interface on a new enumerator object that you can use to iterate through the list of all the shared property groups in the process.

Remarks

You use the **get_NewEnum** method to obtain a reference to an enumerator object. You should immediately cast the returned value to the **IEnumVariant** interface. This interface exposes several methods you can use to iterate through a list of string expressions representing shared property group names. Once you have a name, you can use the **getGroup** method to obtain a reference to the shared property group it represents. When you enumerate the shared property groups, all groups will be included. However, if you then call **CreatePropertyGroup** to add a new group, the existing enumerator won't include the new group even if you call **Reset** or **Clone**. To include the new group, you must create a new enumerator by calling **NewEnum** again.

Example

See Also

ISharedPropertyGroupManager.getGroup Method, **ISharedPropertyGroup Interface**

get_NewEnum Method Example

```
import com.ms.mtx.*;

ISharedPropertyGroupManager spgm = null;
IEnumVariant myEnum = null;
Variant v;
int i;

// Get the enumerator object and
// cast the returned interface to IEnumVariant.
myEnum = (IEnumVariant)spgm.get_NewEnum();

// Use the enumerator to iterate through
// the property group names.
for(i = 0; i < 10; i++){
   v = Enum.Next(1);
   // Do something with the returned
   // property group names.
}
```

ISharedPropertyGroup Interface

The **ISharedPropertyGroup** interface is used to create and access the shared properties in a shared property group.

Remarks

The **ISharedPropertyGroup** interface is declared in the package `com.ms.mtx.`

You can access the **ISharedPropertyGroup** interface by creating a **SharedPropertyGroup** object with the **ISharedPropertyGroupManager.CreatePropertyGroup** method.

The **ISharedPropertyGroup** interface exposes the following methods.

CreateProperty	Creates a new shared property identified by a string expression that's unique within its property group.
CreatePropertyByPosition	Creates a new shared property identified by a numeric index within its property group.
getProperty	Returns a reference to a shared property, given the string name by which the property is identified.
getPropertyByPosition	Returns a reference to a shared property, given its numeric index in the shared property group.

See Also

Sharing State, **ISharedPropertyGroupManager** Interface

ISharedPropertyGroup.CreateProperty Method

Creates and returns a reference to a new **SharedProperty** with a specified name. If a shared property by that name already exists, **CreateProperty** returns a reference to the existing property.

Provided By

ISharedPropertyGroup Interface

ISharedProperty CreateProperty (
  String *name*,
  boolean[] *exists;*
);

Parameters

name
  [in] The name of the property to create. You can use this name later to obtain a reference to this property by using the **getProperty** method.

exists
  [out] An array of one Boolean value that's set to `true` on return from this method if the shared property specified in the *name* parameter existed prior to this call, and `false` if the property was created by this call.

Return Value

A reference to a shared property object with the name specified in the *name* parameter, or `null` if an error is encountered.

Remarks

When you create a shared property, its value is set to the default, which is a Variant with an integer value of 0.

If you create a shared property with the **CreateProperty** method, you can access that property only by using the **getProperty** method. You can't assign a numeric index to the same property and then access it by using the **getPropertyByPosition** method.

The same shared property group can contain some shared property objects that are identified by name and others that are identified by position.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup.CreatePropertyByPosition Method

Creates a new shared property identified by a numeric index that's unique within the property group. If a shared property with the specified index already exists, **CreatePropertyByPosition** returns a reference to the existing one.

Provided By

ISharedPropertyGroup Interface

ISharedProperty CreatePropertyByPosition (
 int *index*,
 boolean[] *exists;*
);

Parameters

index
 [in] The numeric index within the **SharedPropertyGroup** by which the new property will be referenced. You can use this index later to retrieve the shared property with the **getPropertyByPosition** method.

exists
 [out] An array of one boolean value. If *exists* is set to `true` on return from this method, the shared property specified by *index* existed prior to this call. If it's set to `false`, the property was created by this call.

Return Value

A reference to a shared property object that's identified by the numeric index passed in the *index* parameter, or `null` if an error is encountered.

Remarks

When you create a shared property, its value is set to the default, which is a Variant with an integer value of 0.

If you create a **SharedProperty** object with the **CreatePropertyByPosition** method, you can access that property only by using the **getPropertyByPosition** method. You can't assign a string name to the same property and then access it by using the **getProperty** method. Accessing a property by position is faster than accessing a property by using a string name because it requires less overhead.

The same shared property group can contain some **SharedProperty** objects that are identified by position and others that are identified by name.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup.CreatePropertyByPosition Method Example

```
import com.ms.mtx.*;


ISharedPropertyGroupManager propGpMgr = null;
ISharedPropertyGroup propGp = null;
ISharedProperty propNextNum = null;
Variant vNext;
int iNextValue = 0;
boolean[] afAlreadyExists = new boolean[1];
int[] aiIsolationMode = new int[1];
aiIsolationMode[0] =
   ISharedPropertyGroupManager.LOCKMODE_SETGET;
int[] aiReleaseMode = new int[1];
aiReleaseMode[0] =
   ISharedPropertyGroupManager.RELEASEMODE_PROCESS;

// Create the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
propGpMgr = new SharedPropertyGroupManager();
propGp = propGpMgr.CreatePropertyGroup
   ("Counter", aiIsolationMode,
   aiReleaseMode, afAlreadyExists);
propNextNum = propGp.CreatePropertyByPosition
   (0, fAlreadyExists);

// Get the next number and increment it.
VariantInit(&vNext);
vNext = propNextNum.getValue();
iNextValue = vNext.getInt();
vNext.putInt(iNextValue + 1);
propNextNum.putValue(vNext);
```

ISharedPropertyGroup.getProperty Method

Returns a reference to an existing shared property identified by a string name.

Provided By

ISharedPropertyGroup Interface

ISharedProperty getProperty (
String *name*,
);

Parameters

name
[in] A string expression that contains the name of the shared property to retrieve.

Return Value

A reference to the shared property specified in the *name* parameter, or `null` if the property doesn't exist.

Remarks

You can use only the **getProperty** method to access properties that were created with the **CreateProperty** method. To access properties that were created with the **CreatePropertyByPosition** method, use the **getPropertyByPosition** method.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_PropertyByPosition** Method

ISharedPropertyGroupManager.getGroup, ISharedPropertyGroup.getProperty Methods Example

```
import com.ms.mtx.*;


ISharedPropertyGroupManager propGpMgr = null;
ISharedPropertyGroup propGp = null;
ISharedProperty propNextNum = null;
Variant vNext;
int iNextValue = 0;

// Get the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
propGpMgr = new SharedPropertyGroupManager();
propGp = propGpMgr.getGroup("Counter");
propNextNum = propGp.getProperty("NextNum");

// Get the next number and increment it.
VariantInit(&vNext);
vNext = propNextNum.getValue();
iNextValue = vNext.getInt();
vNext.putInt(iNextValue + 1);
propNextNum.putValue(vNext);
```

ISharedPropertyGroup.getPropertyByPosition Method

Returns a reference to an existing shared property identified by its numeric index within the property group.

Provided By

ISharedPropertyGroup Interface

ISharedProperty getPropertyByPosition (
    INT *index*,
);

Parameters

index
    [in] The numeric index within the **SharedPropertyGroup** of the property to retrieve.

Return Value

A reference to the shared property specified by the *index* parameter, or `null` if the property doesn't exist.

Remarks

You can use only the **getPropertyByPosition** method to access properties that were created with the **CreatePropertyByPosition** method. To access properties that were created with the **CreateProperty** method, use the **getProperty** method.

Example

See Also

Sharing State, **ISharedPropertyGroup::CreateProperty** Method, **ISharedPropertyGroup::CreatePropertyByPosition** Method, **ISharedPropertyGroup::get_Property** Method

ISharedPropertyGroup.getPropertyByPosition Method Example

```
import com.ms.mtx.*;


ISharedPropertyGroupManager propGpMgr = null;
ISharedPropertyGroup propGp = null;
ISharedProperty propNextNum = null;
Variant vNext;
int iNextValue = 0;

// Get the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
propGpMgr = new SharedPropertyGroupManager();
propGp = propGpMgr.getGroup("Counter");
propNextNum = propGp.getPropertyByPosition(0);

// Get the next number and increment it.
VariantInit(&vNext);
vNext = propNextNum.getValue();
iNextValue = vNext.getInt();
vNext.putInt(iNextValue + 1);
propNextNum.putValue(vNext);
```

ISharedProperty Interface

The **ISharedProperty** interface is used to set or retrieve the value of a shared property. A shared property can contain any data type that can be represented by a variant.

Remarks

The **ISharedProperty** interface is declared in the package `com.ms.mtx.`

You can access the **ISharedProperty** interface by creating a **SharedProperty** object with the **ISharedPropertyGroup.CreateProperty** method or the **ISharedPropertyGroup.CreatePropertyByPosition** method.

A **SharedProperty** object can be created or accessed only from within a **SharedPropertyGroup**.

The **ISharedProperty** interface exposes the following methods.

Method	Description
getValue	Retrieves the value of a shared property.
putValue	Sets the value of a shared property.

See Also

Sharing State, MTS Supported Variant Types

ISharedProperty.getValue Method

Retrieves the value of a shared property.

Provided By

ISharedProperty Interface

Variant getValue ( );

Return Value

A Variant in which the value of the shared property will be returned.

Example

See Also

MTS Supported Variant Types, Sharing State

ISharedProperty.putValue Method

Assigns a value to a shared property.

Provided By

ISharedProperty Interface

void putValue (
  Variant *value*
);

Parameters

value
  [in] A Variant containing the value to assign to the **SharedProperty** object.

Example

See Also

MTS Supported Variant Types, Sharing State

CreatePropertyGroup, CreateProperty, putValue, getValue Methods Example

```
import com.ms.mtx.*;


ISharedPropertyGroupManager propGpMgr = null;
ISharedPropertyGroup propGp = null;
ISharedProperty propNextNum = null;
Variant vNext;
int iNextValue = 0;
boolean[] afAlreadyExists = new boolean[1];
int[] aiIsolationMode = new int[1];
aiIsolationMode[0] =
   ISharedPropertyGroupManager.LOCKMODE_SETGET;
int[] aiReleaseMode = new int[1];
aiReleaseMode[0] =
   ISharedPropertyGroupManager.RELEASEMODE_PROCESS;

// Create the SharedPropertyGroupManager,
// SharedPropertyGroup, and SharedProperty.
propGpMgr = new SharedPropertyGroupManager();
propGp = propGpMgr.CreatePropertyGroup
   ("Counter", aiIsolationMode,
   aiReleaseMode, afAlreadyExists);
propNextNum = propGp.CreateProperty
   ("NextNum", afAlreadyExists);

// Get the next number and increment it.
VariantInit(&vNext);
vNext = propNextNum.getValue();
iNextValue = vNext.getInt();
vNext.putInt(iNextValue + 1);
propNextNum.putValue(vNext);
```

TransactionContext Object

The **TransactionContext** object is used by a base client to compose the work of one or more MTS objects into an atomic transaction and to commit or abort the transaction.

Remarks

To use the **TransactionContext** object, you must set a reference to the Transaction Context Type Library (txctx.dll).

You can use a **TransactionContext** object to scope a transaction from a base client. You begin the transaction by instantiating a **TransactionContext** object, and you end the transaction by calling **Commit** or **Abort** on the object. The base client itself never executes within the transaction.

The **TransactionContext** component is a standard MTS component. The component's transaction attribute is set to **Requires a new transaction**, which means that a **TransactionContext** object is always the root of a transaction. When a base client instantiates an object by using the **TransactionContext** object's **CreateInstance** method, the new object and its descendants will participate in the**TransactionContext** object's transaction unless the new object's transaction attribute is set to **Requires a new transaction** or **Does not support transactions**.

You could easily write your own **TransactionContext** component. You would simply create a component that implements the methods **Commit**, **Abort**, and **CreateInstance**, and set the component's transaction attribute to **Requires a new transaction**. The three methods would do nothing more than call **GetObjectContext** and invoke their **ObjectContext** object's **SetComplete**, **SetAbort**, and **CreateInstance** methods, respectively.

Before you use **TransactionContext** to compose the work of existing components in a transaction, you should consider implementing a separate component that not only composes their work but encapsulates it into a reusable unit. This new component would not only serve the needs of the current base client, but other clients could also use it. In one approach, the base client instantiates a **TransactionContext** object, calls its **CreateInstance** method to instantiate other objects, calls various methods on those objects, and finally calls **Commit** or **Abort** on the **TransactionContext** object. In the other approach, you create a new component that requires a transaction. This new component instantiates the other objects using its **ObjectContext** object's **CreateInstance** method, calls the relevant methods on those other objects itself, and then calls **SetComplete** or **SetAbort** on its **ObjectContext** when it's done. Using this approach, the base client only needs to instantiate this one object, and invoke one method on it, and the object does the rest of the work. When other clients require the same functionality, they can reuse the new component.

You obtain a reference to a **TransactionContext** object with **CreateObject**. For example:

```
Set objTransactionContext = _
   CreateObject("TxCtx.TransactionContext")
```

The **TransactionContext** object provides the following methods.

Method	Description
Abort	Aborts the work of all MTS objects participating in the current transaction. The transaction is completed on return from this method.
Commit	Attempts to commit the work of all MTS objects participating in the current transaction. If any of the MTS objects participating in the transaction have called **SetAbort** or **DisableCommit**, or if a system error has occurred, the transaction will be aborted. Otherwise, the transaction will be committed. In either case, the transaction is completed on return from this method.
CreateInstance	Instantiates another MTS object. If the component that provides the object is configured to support or require a transaction, then the

new object runs under the transaction of the **TransactionContext**object.

See Also

Transaction Context Objects, Base Clients, Transactions

Abort Method

Aborts the current transaction.

Applies To

TransactionContext Object

Syntax

transactioncontextobject.**Abort**

The *transactioncontextobject* placeholder represents an object variable that evaluates to a **TransactionContext** object.

Remarks

When a base client calls **Abort**, all objects that participated in the transaction are automatically deactivated. Any database updates made by those objects are rolled back. The transaction is completed on return from this method. If another call is made on the **TransactionContext** object after the **TransactionContext** object has returned from a call in which it called the **Abort** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetAbort**

Abort, Commit Methods Example

```
Dim objTxCtx As TransactionContext
Dim objMyObject As MyCompany.MyObject
Dim userCanceled As Boolean

' Get TransactionContext.
Set objTxCtx = _
   CreateObject("TxCtx.TransactionContext")

' Create an instance of some component.
Set objMyObject= _
   objTxCtx.CreateInstance("MyCompany.MyObject")

' Do some work here.

' If something goes wrong, abort the transaction.
If userCanceled Then
   objTxCtx.Abort

' Otherwise, commit it.
Else
   objTxCtx.Commit
End If
```

Commit Method

Attempts to commit the current transaction.

Applies To

TransactionContext Object

Syntax

transactioncontextobject.**Commit**

The *transactioncontextobject* placeholder represents an object variable that evaluates to a **TransactionContext** object.

Remarks

Calling **Commit** doesn't guarantee that a transaction will be committed. If any MTS object that was part of the transaction has returned from a method after calling **SetAbort**, the transaction will be aborted. If any object that was part of the transaction has called **DisableCommit** and hasn't yet called **EnableCommit** or **SetComplete**, the transaction will also be aborted. Any error that causes Microsoft Distributed Transaction Coordinator to abort a transaction will also abort an MTS transaction.

When a base client calls **Commit**, regardless of whether the transaction commits or aborts, the transaction is completed on return from this method and all objects that participated in the transaction are automatically deactivated. If another call comes in after the **TransactionContext** object has returned from a call in which it called the **Commit** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetComplete**

CreateInstance Method

Instantiates an MTS object that will execute within the scope of the transaction that was initiated with the creation of the **TransactionContext** object.

Applies To

TransactionContext Object

Syntax

Set *object* = *transactioncontextobject*.**CreateInstance**(*programmaticID*)

Part

object
An object variable that evaluates to an MTS object.

transactioncontextobject
An object variable that represents the **TransactionContext** object from which to create the new object.

programmaticID
The programmatic Id of the new object's component.

Remarks

When a base client uses the **TransactionContext** object's **CreateInstance** method to instantiate an MTS object, the new object executes within the transaction context object's activity. If the transaction attribute of the new object's component is set to either **Supports transactions** or **Requires a transaction**, the new object also inherits the transaction initiated with the creation of the **TransactionContext** object. However, if the component that provides the new object has its transaction attribute set to **Does not support transactions**, the object neither inherits the transaction nor passes it on to objects it subsequently creates. If the component that provides the new object has its transaction attribute set to **Requires a new transaction**, the MTS run-time environment initiates a new transaction for the new object, and that transaction is the one that's inherited by objects it subsequently creates.

In this respect, using **CreateInstance** is comparable to using **CoCreateInstance** and specifying NULL for the controlling **IUnknown** interface (*pUnkOuter*).

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **CreateInstance**

CreateInstance Method, TransactionContext Object Example

```
Dim objTxCtx As TransactionContext
Dim objMyObject As MyCompany.MyObject

' Get TransactionContext.
Set objTxCtx = _
   CreateObject("TxCtx.TransactionContext")

' Create an instance of MyObject.
Set objMyObject= _
   objTxCtx.CreateInstance("MyCompany.MyObject")
```

ITransactionContextEx Interface

The **ITransactionContextEx** interface is used by a base client to compose the work of one or more MTS objects into an atomic transaction and to commit or abort the transaction.

Remarks

The header file for the **ITransactionContextEx** interface is txctx.h. You must also link mtxguid.lib to your project to use this interface.

You can use a **TransactionContextEx** object to scope a transaction from a base client. You begin the transaction by instantiating a **TransactionContextEx** object, and you end the transaction by calling **Commit** or **Abort** on the object. The base client itself never executes within the transaction.

The **TransactionContextEx** component is a standard MTS component. The component's transaction attribute is set to **Requires a new transaction**, which means that a **TransactionContextEx** object is always the root of a transaction. When a base client instantiates an object by using the **ITransactionContextEx::CreateInstance** method, the new object and its descendants will participate in the**TransactionContextEx** object's transaction unless the new object's transaction attribute is set to **Requires a new transaction** or **Does not support transactions**.

You could easily write your own **TransactionContextEx** component. You would simply create a component that implements the methods **Commit**, **Abort**, and **CreateInstance**, and set the component's transaction attribute to **Requires a new transaction**. The three methods would do nothing more than call **GetObjectContext** and invoke their **ObjectContext** object's **SetComplete**, **SetAbort**, and **CreateInstance** methods, respectively.

Before you use **TransactionContextEx** to compose the work of existing components in a transaction, you should consider implementing a separate component that not only composes their work but encapsulates it into a reusable unit. This new component would not only serve the needs of the current base client, but other clients could also use it. In one approach, the base client instantiates a **TransactionContextEx** object, calls its **CreateInstance** method to instantiate other objects, calls various methods on those objects, and finally calls **Commit** or **Abort** on the **TransactionContextEx** object. In the other approach, you create a new component that requires a transaction. This new component instantiates the other objects using its **ObjectContext** object's **CreateInstance** method, calls the relevant methods on those other objects itself, and then calls **SetComplete** or **SetAbort** on its **ObjectContext** when it's done. Using this approach, the base client only needs to instantiate this one object, and invoke one method on it, and the object does the rest of the work. When other clients require the same functionality, they can reuse the new component.

You obtain a reference to the **ITransactionContextEx** interface by creating a **TransactionContextEx** object with a call to **CoCreateInstance**. For example:

```
CoCreateInstance(CLSID_TransactionContextEx, NULL, CLSCTX_INPROC,
IID_ITransactionContextEx, (void**)&m_pTransactionContext);
```

The **ITransactionContextEx** interface exposes the following methods.

Method	Description
Abort	Aborts the work of all MTS objects participating in the current transaction. The transaction is completed on return from this method.
Commit	Attempts to commit the work of all MTS objects participating in the current transaction. If any of the MTS objects participating in the transaction have called **SetAbort** or **DisableCommit**, or if a system error has occurred, the transaction will be aborted. Otherwise, the transaction will be committed. In either case, the transaction is completed on return from this method.
CreateInstance	Instantiates another MTS object. If the component that provides the

object is configured to support or require a transaction, then the new object runs under the transaction of the **TransactionContextEx**object.

See Also

Transaction Context Objects, Base Clients, Transactions

ITransactionContextEx::Abort Method

Aborts the current transaction.

Provided By

ITransactionContextEx Interface

HRESULT ITransactionContextEx::Abort ( );

Return Values

S_OK
The transaction was aborted.

E_FAIL
The **TransactionContextEx** object isn't running under a MTS process. This could happen if the **TransactionContextEx** component's Registry entry has been corrupted.

E_UNEXPECTED
An unexpected error occurred.

Remarks

When a base client calls **Abort**, all objects that participated in the transaction are automatically deactivated. Any database updates made by those objects are rolled back. The transaction is completed on return from this method. If another call is made on the **TransactionContextEx** object after the **TransactionContextEx** object has returned from a call in which it called the **Abort** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetAbort**

ITransactionContextEx::Abort, ITransactionContextEx::Commit Methods Example

```
#include <Txctx.h>

ITransactionContextEx* pTransactionContext = NULL;
IMyObject* pMyObject = NULL;
boolean bUserCanceled = FALSE;
HRESULT hr;

// Get TransactionContextEx.
hr = CoCreateInstance(CLSID_ITransactionContextEx,
   NULL, CLSCTX_INPROC, IID_ITransactionContextEx,
   (void**) &pTransactionContext);

// Create an instance of MyObject.
hr = pTransactionContext->CreateInstance
   (CLSID_CMyObject, IID_IMyObject,
   (void**) &pMyObject);

// Do some work here.

// If something goes wrong, abort the transaction.
if (bUserCanceled)
   pTransactionContext->Abort();

// Otherwise, commit it.
else
   pTransactionContext->Commit();
```

ITransactionContextEx::Commit Method

Attempts to commit the current transaction.

Provided By

ITransactionContextEx Interface

HRESULT ITransactionContextEx::Commit ( );

Return Values

S_OK
: The transaction was committed.

E_FAIL
: The **TransactionContextEx** object isn't running under a MTS process. This could happen if the **TransactionContextEx** component's Registry entry has been corrupted.

E_UNEXPECTED
: An unexpected error occurred.

CONTEXT_E_ABORTED
: The transaction was aborted.

Remarks

Calling **Commit** doesn't guarantee that a transaction will be committed. If any MTS object that was part of the transaction has returned from a method after calling **SetAbort**, the transaction will be aborted. If any object that was part of the transaction has called **DisableCommit** and hasn't yet called **EnableCommit** or **SetComplete**, the transaction will also be aborted. Any error that causes Microsoft Distributed Transaction Coordinator to abort a transaction will also abort an MTS transaction.

When a base client calls **Commit**, regardless of whether the transaction commits or aborts, the transaction is completed on return from this method and all objects that participated in the transaction are automatically deactivated. If another call comes in after the **TransactionContextEx** object has returned from a call in which it called the **Commit** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetComplete**

ITransactionContextEx::CreateInstance Method

Instantiates an MTS object that will execute within the scope of the transaction that was initiated with the creation of the **TransactionContextEx** object.

Provided By

ITransactionContextEx Interface

HRESULT ITransactionContextEx::CreateInstance (
    REFCLSID *rclsid*,
    REFIID *riid*,
    LPVOID FAR* *ppvObj*
);

Parameter

rclsid
    [in] A reference to the CLSID of the type of object to instantiate.

riid
    [in] A reference to the interface ID of the interface through which you want to communicate with the new object.

ppvObj
    [out] A reference to a new object of the type specified by the *rclsid* argument, through the interface specified by the *riid* argument.

Return Values

S_OK
    A reference to the object is returned in the *ppvObj* parameter.

REGDB_E_CLASSNOTREG
    The component specified by *rclsid* is not registered as a COM component.

E_OUTOFMEMORY
    There's not enough memory available to instantiate the object.

E_INVALIDARG
    The argument passed in the *ppvObj* parameter is invalid.

E_UNEXPECTED
    An unexpected error occurred.

Remarks

When a base client uses the **ITransactionContextEx::CreateInstance** method to instantiate an MTS object, the new object executes within the transaction context object's activity. If the transaction attribute of the new object's component is set to either **Supports transactions** or **Requires a transaction**, the new object also inherits the transaction initiated with the creation of the **TransactionContextEx** object. However, if the component that provides the new object has its transaction attribute set to **Does not support transactions**, the object neither inherits the transaction nor passes it on to objects it subsequently creates. If the component that provides the new object has its transaction attribute set to **Requires a new transaction**, the MTS run-time environment initiates a new transaction for the new object, and that transaction is the one that's inherited by objects it subsequently creates.

If the Microsoft Distributed Transaction Coordinator is not running and the object is transactional, the object is successfully created. However, method calls to that object will fail with CONTEXT_E_TMNOTAVAILABLE. Objects cannot recover from this condition and should be released.

MTS always uses standard marshaling. Even if a component exposes the **IMarshal** interface, its **IMarshal** methods will never be called by the MTS run-time environment.

Note You can't create MTS objects as part of an aggregation. In this respect, using **CreateInstance** is comparable to using **CoCreateInstance** and specifying NULL for the controlling **IUnknown** interface (*pUnkOuter*).

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **CreateInstance**

ITransactionContextEx::CreateInstance Method Example

```
#include <Txctx.h>

ITransactionContextEx* pTransactionContext = NULL;
IMyObject* pMyObject = NULL;
HRESULT hr;

// Get TransactionContextEx.
hr = CoCreateInstance(CLSID_ITransactionContextEx,
   NULL, CLSCTX_INPROC, IID_ITransactionContextEx,
   (void**) &pTransactionContext);

// Create an instance of MyObject.
hr = pTransactionContext->CreateInstance
   (CLSID_CMyObject, IID_IMyObject,
   (void**) &pMyObject);
```

ITransactionContextEx Interface

The **ITransactionContextEx** interface is used by a base client to compose the work of one or more MTS objects into an atomic transaction and to commit or abort the transaction.

Remarks

The **ITransactionContextEx** interface is declared in the package `com.ms.mtx.`

You can use a **TransactionContextEx** object to scope a transaction from a base client. You begin the transaction by instantiating a **TransactionContextEx** object, and you end the transaction by calling **Commit** or **Abort** on the object. The base client itself never executes within the transaction.

The **TransactionContextEx** component is a standard MTS component. The component's transaction attribute is set to **Requires a new transaction**, which means that a **TransactionContextEx** object is always the root of a transaction. When a base client instantiates an object by using the **ITransactionContextEx.CreateInstance** method, the new object and its descendants will participate in the**TransactionContextEx** object's transaction unless the new object's transaction attribute is set to **Requires a new transaction** or **Does not support transactions**.

You could easily write your own **TransactionContextEx** component. You would simply create a component that implements the methods **Commit**, **Abort**, and **CreateInstance**, and set the component's transaction attribute to **Requires a new transaction**. The three methods would do nothing more than call **GetObjectContext** and invoke their **ObjectContext** object's **SetComplete**, **SetAbort**, and **CreateInstance** methods, respectively.

Before you use **TransactionContextEx** to compose the work of existing components in a transaction, you should consider implementing a separate component that not only composes their work but encapsulates it into a reusable unit. This new component would not only serve the needs of the current base client, but other clients could also use it. In one approach, the base client instantiates a **TransactionContextEx** object, calls its **CreateInstance** method to instantiate other objects, calls various methods on those objects, and finally calls **Commit** or **Abort** on the **TransactionContextEx** object. In the other approach, you create a new component that requires a transaction. This new component instantiates the other objects using its **ObjectContext** object's **CreateInstance** method, calls the relevant methods on those other objects itself, and then calls **SetComplete** or **SetAbort** on its **ObjectContext** when it's done. Using this approach, the base client only needs to instantiate this one object, and invoke one method on it, and the object does the rest of the work. When other clients require the same functionality, they can reuse the new component.

You obtain a reference to the **ITransactionContextEx** interface by creating a **TransactionContextEx** object. For example:

```
new TransactionContextEx();
```

The **ITransactionContextEx** interface exposes the following methods.

Method	Description
Abort	Aborts the work of all MTS objects participating in the current transaction. The transaction is completed on return from this method.
Commit	Attempts to commit the work of all MTS objects participating in the current transaction. If any of the MTS objects participating in the transaction have called **SetAbort** or **DisableCommit**, or if a system error has occurred, the transaction will be aborted. Otherwise, the transaction will be committed. In either case, the transaction is completed on return from this method.
CreateInstance	Instantiates another MTS object. If the component that provides the object is configured to support or require a transaction, then the new object runs under the transaction of the

TransactionContextEx object.

See Also

Transaction Context Objects, Base Clients, Transactions

ITransactionContextEx.Abort Method

Aborts the current transaction.

Provided By

ITransactionContextEx Interface

void Abort ( );

Remarks

When a base client calls **Abort**, all objects that participated in the transaction are automatically deactivated. Any database updates made by those objects are rolled back. The transaction is completed on return from this method. If another call is made on the **TransactionContextEx** object after the **TransactionContextEx** object has returned from a call in which it called the **Abort** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetAbort**

ITransactionContext.Abort, ITransactionContext.Commit Methods Example

```
import com.ms.mtx.*;

ITransactionContextEx myTransactionContext = null;
IMyObject myObject = null;
boolean userCanceled = false;

// Get TransactionContextEx.
myTransactionContext = new TransactionContextEx();

// Create an instance of MyObject.
myObject = myTransactionContext.CreateInstance (CMyObject.clsid,
IMyObject.iid);

// Do some work here.

// If something goes wrong, abort the transaction.
if (userCanceled)
   myTransactionContext.Abort();

// Otherwise, commit it.
else
   pTransactionContext.Commit();
```

ITransactionContextEx.Commit Method

Attempts to commit the current transaction.

Provided By

ITransactionContextEx Interface

void Commit ( );

Remarks

Calling **Commit** doesn't guarantee that a transaction will be committed. If any MTS object that was part of the transaction has returned from a method after calling **SetAbort**, the transaction will be aborted. If any object that was part of the transaction has called **DisableCommit** and hasn't yet called **EnableCommit** or **SetComplete**, the transaction will also be aborted. Any error that causes Microsoft Distributed Transaction Coordinator to abort a transaction will also abort an MTS transaction.

When a base client calls **Commit**, regardless of whether the transaction commits or aborts, the transaction is completed on return from this method and all objects that participated in the transaction are automatically deactivated. If another call comes in after the **TransactionContextEx** object has returned from a call in which it called the **Commit** method, a new transaction is started.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **SetComplete**

ITransactionContextEx.CreateInstance Method

Instantiates an MTS object that will execute within the scope of the transaction that was initiated with the creation of the **TransactionContextEx** object.

Provided By

ITransactionContextEx Interface

IUnknown CreateInstance (
  _Guid *clsid*,
  _Guid *iid*,
);

Parameter

clsid
  [in] A reference to the CLSID of the type of object to instantiate.
iid
  [in] Any interface that's implemented by the object you want to instantiate.

Return Value

A reference to the **IUnknown** interface on a new instance of the MTS component specified in the *clsid* parameter.

Remarks

When a base client uses the **ITransactionContextEx.CreateInstance** method to instantiate an MTS object, the new object executes within the transaction context object's activity. If the transaction attribute of the new object's component is set to either **Supports transactions** or **Requires a transaction**, the new object also inherits the transaction initiated with the creation of the **TransactionContextEx** object. However, if the component that provides the new object has its transaction attribute set to **Does not support transactions**, the object neither inherits the transaction nor passes it on to objects it subsequently creates. If the component that provides the new object has its transaction attribute set to **Requires a new transaction**, the MTS run-time environment initiates a new transaction for the new object, and that transaction is the one that's inherited by objects it subsequently creates.

CreateInstance always returns the **IUnknown** interface on the newly instantiated object. You should immediately cast the returned value to the interface with which you want to communicate with the new object. The interface ID you pass in the *iid* parameter doesn't have to be the same interface to which you cast the returned value, but it must be an interface that's implemented by the object you want to instantiate.

MTS always uses standard marshaling. Even if a component exposes the **IMarshal** interface, its **IMarshal** methods will never be called by the MTS run-time environment.

Note You can't create MTS objects as part of an aggregation.

Example

See Also

Transaction Context Objects, Base Clients, Transactions, **CreateInstance**

ITransactionContext.CreateInstance Method Example

```
import com.ms.mtx.*;

ITransactionContextEx myTransactionContext = null;
IMyObject myObject = null;

// Get TransactionContextEx.
myTransactionContext = new TransactionContextEx();

// Create an instance of MyObject.
myObject = (IMyObject)
   myTransactionContext.CreateInstance
   (CMyObject.clsid, IMyObject.iid);
```

SecurityProperty Object

The **SecurityProperty** object is used to determine the current object's caller or creator.

Remarks

To use the **SecurityProperty** object, you must set a reference to Microsoft Transaction Server Type Library (mtxas.dll).

You obtain a reference to an object's **SecurityProperty** object by calling **Security** on the object's **ObjectContext**. For example:

```
Set secObject = ctxObject.Security
```

The **SecurityProperty** object provides the following methods.

Method	Description
GetDirectCallerName	Retrieves the user name associated with the external process that called the currently executing method.
GetDirectCreatorName	Retrieves the user name associated with the external process that directly created the current object.
GetOriginalCallerName	Retrieves the user name associated with the base process that initiated the call sequence from which the current method was called.
GetOriginalCreatorName	Retrieves the user name associated with the base process that initiated the activity in which the current object is executing.

See Also

Programmatic Security, Advanced Security Methods, **ObjectContext** Object

GetDirectCallerName Method

Retrieves the user name associated with the external process that called the currently executing method.

Applies To

SecurityProperty Object

Syntax

username = *securityproperty*.**GetDirectCallerName( )**

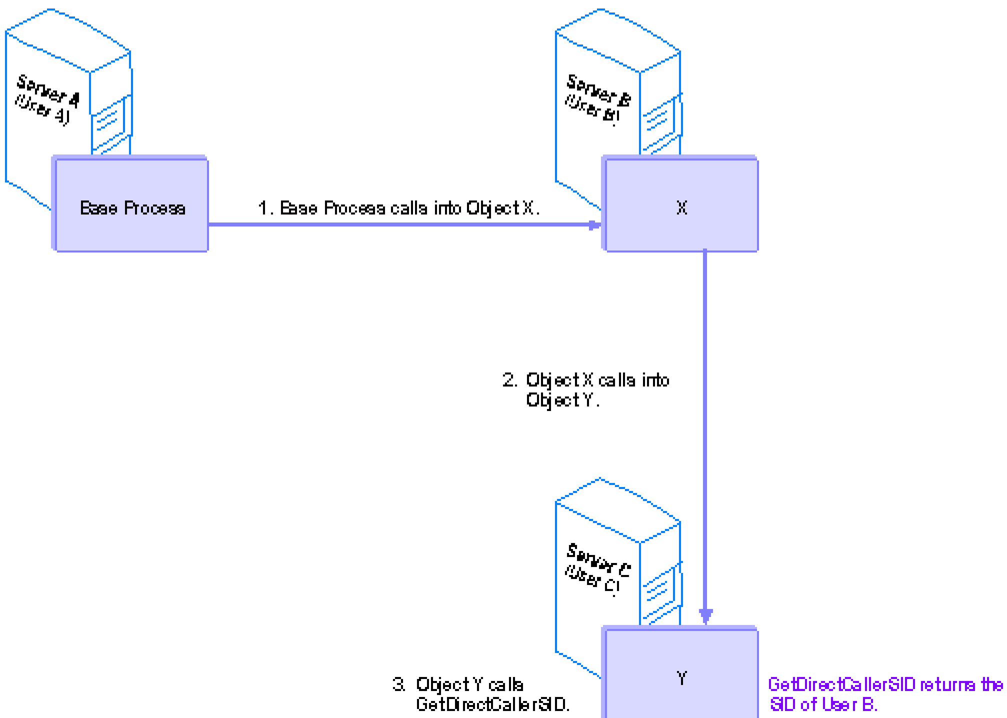
Part

username
The user name associated with the process from which the current method was invoked.

securityproperty
An object variable that evaluates to a **SecurityProperty** object.

Remarks

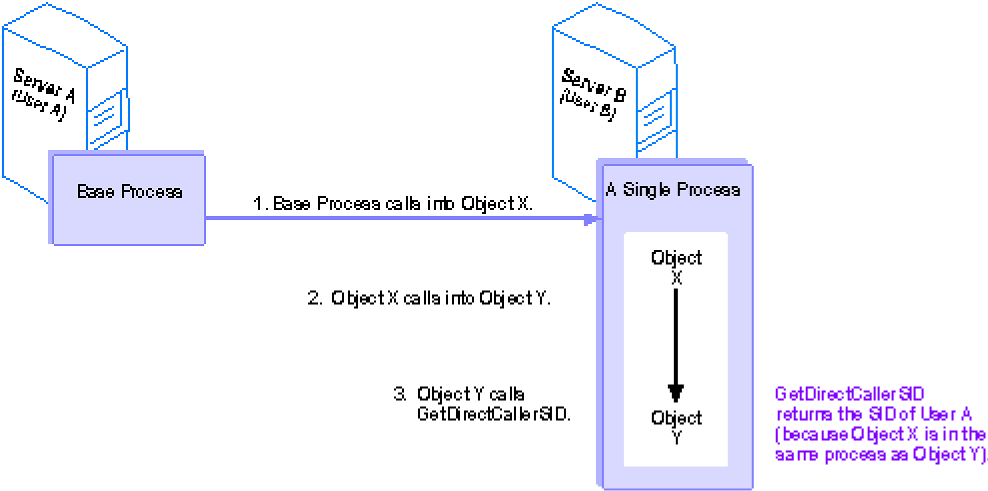
You use the **GetDirectCallerName** method to determine the user name associated with the process that called the object's currently executing method. The following scenarios illustrate the functionality of the **GetDirectCallerName** method.



A base process running on server A, as user A, calls into object X on server B, running as user B. Then object X calls into object Y, running on server C. If object Y calls **GetDirectCallerName**, the name of user B is returned.

Security can only be enforced across process boundaries. This means that the name returned by

GetDirectCallerName is the name associated with the process that called into the process in which the current object is running, not necessarily the immediate caller into the object itself. If an object calls into another object within the same process, when the second object calls **GetDirectCallerName**, it will get the name of the most immediate caller outside its own process boundary, not the name of the object that directly called into it.



A base process, running on server A as user A, calls into object X on server B, running as user B. Then object X calls into object Y, running in the same process as object X, also on server B. When object Y calls **GetDirectCallerName**, the name of user A is returned , not the name of user B.

Example

See Also

Programmatic Security, Advanced Security Methods, **ObjectContext** Object

GetDirectCallerName Method Example

```
Public Function ComponentDirectCaller() As String

    Dim objCtx As ObjectContext

    Set objCtx = GetObjectContext()
    ComponentDirectCaller = _
        objCtx.Security.GetDirectCallerName()

End Function
```

GetDirectCreatorName Method

Retrieves the user name associated with the current object's immediate (out-of-process) creator.

Applies To

SecurityProperty Object

Syntax

username = *securityproperty*.**GetDirectCreatorName( )**

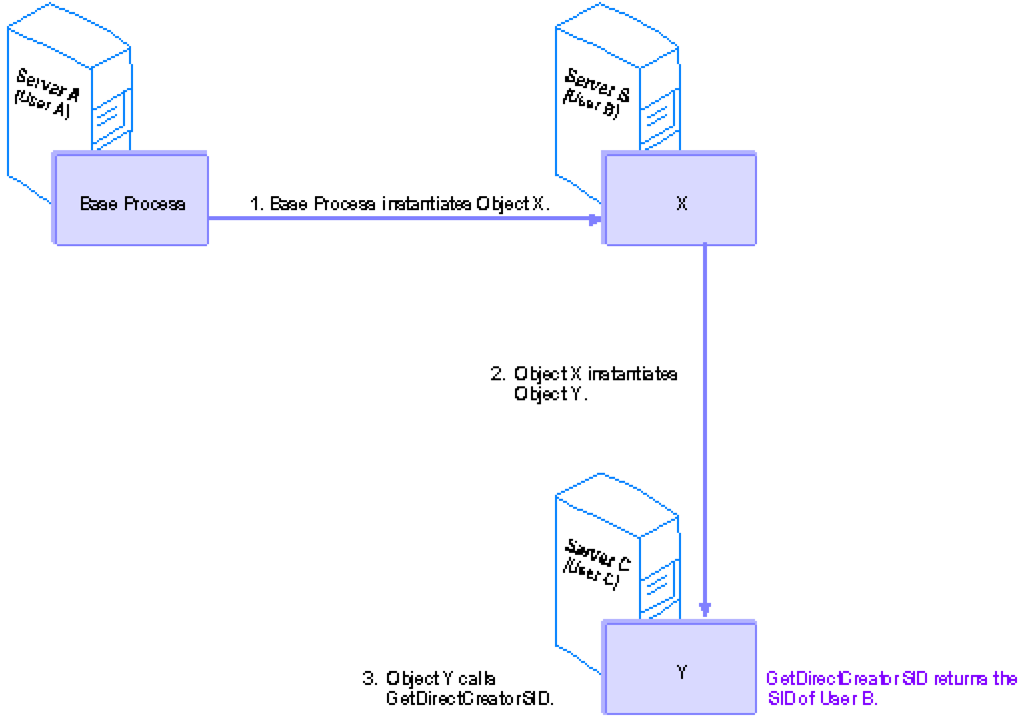
Part

username
The user name associated with the process that directly created the current object.

securityproperty
An object variable that evaluates to a **SecurityProperty** object.

Remarks

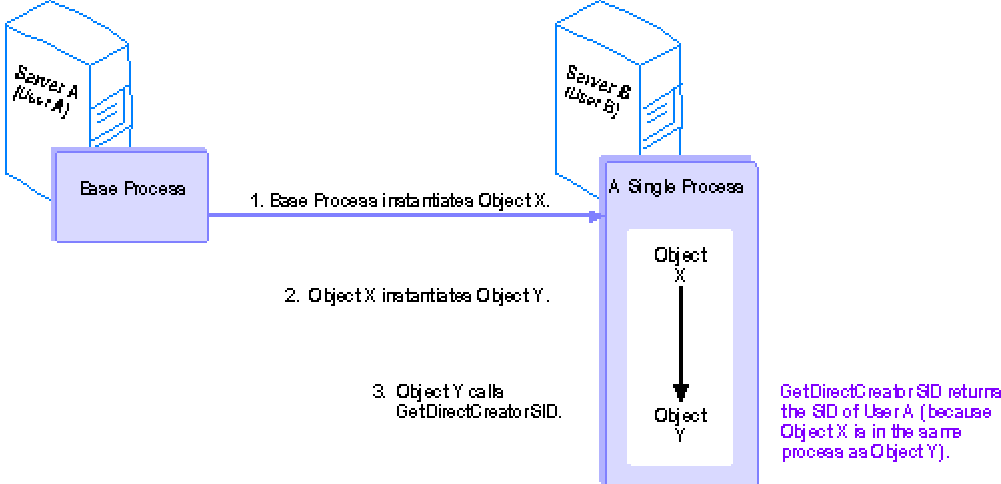
You use the **GetDirectCreatorName** method to determine the user name associated with the process that created the current object. The following scenarios illustrate the functionality of the **GetDirectCreatorName** method.



A base process running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running on server C. If object Y calls **GetDirectCreatorName**, the name of user B is returned.

Security can only be enforced across process boundaries. This means that if an object creates another object within the same process, when the second object calls **GetDirectCreatorName**, it will

get the name of the most immediate creator outside its own process boundary, not the user name associated with the object that actually created it.



A base client running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running in the same process as object X, also on server B. When object Y calls **GetDirectCreatorName**, the name of user A is returned, not the name of user B.

Example

See Also

Programmatic Security, Advanced Security Methods, **ObjectContext** Object

GetDirectCreatorName Method Example

```
Public Function ComponentDirectCreator() As String

    Dim objCtx As ObjectContext

    Set objCtx = GetObjectContext()
    ComponentDirectCreator = _
        objCtx.Security.GetDirectCreatorName()

End Function
```

GetOriginalCallerName Method

Retrieves the user name associated with the base process that initiated the sequence of calls from which the call into the current object originated.

Applies To

SecurityProperty Object

Syntax

username = *securityproperty*.**GetOriginalCallerName( )**

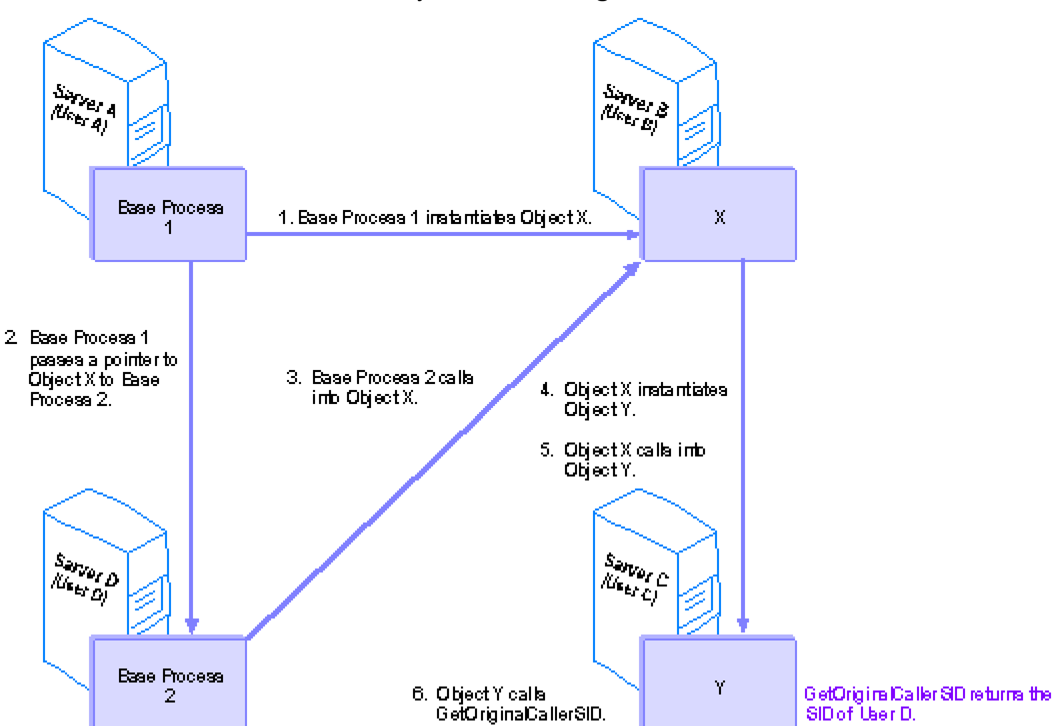
Part

username
The user name associated with the base process that initiated the call sequence from which the current method was called.

securityproperty
An object variable that evaluates to a **SecurityProperty** object.

Remarks

You use the **GetOriginalCallerName** method to determine the user name associated with the original process that initiated the call sequence from which the current method was called. The following scenario illustrates the functionality of the **GetOriginalCallerName** method.



Base process 1, running on server A as user A, creates object X on server B, running as user B. Then base process 1 passes its reference on object X to base process 2, running on server D as user D. Base process 2 uses that reference to call into object X. object X then calls into object Y, running on server C. If object Y then calls **GetOriginalCallerName**, the name of user D is returned.

Note Usually, an object's original caller is the same process as its original creator. The only situation in which the original caller and the original creator would be different is one in which the original creator passes a reference to another process, and the other process initiates the call sequence (as in the preceding example).

Note The path to the original caller is broken if any object along the chain was created by some other means than **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**. For example, if base process 1 uses **CoCreateInstance** to create X, when Y calls **GetOriginalCallerName,** the name it gets back will be the name of user B, not user D. This is because the call sequence is traced back through the objects' context and MTS can only create a context for an object that's created with either **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

Example

See Also

Programmatic Security, Advanced Security Methods, **ObjectContext** Object

GetOriginalCallerName Method Example

```
Public Function ComponentOriginalCaller() As String

    Dim objCtx As ObjectContext

    Set objCtx = GetObjectContext()
    ComponentOriginalCaller = _
        objCtx.Security.GetOriginalCallerName()

End Function
```

GetOriginalCreatorName Method

Retrieves the user name associated with the original base process that initiated the activity in which the current object is executing.

Applies To

SecurityProperty Object

Syntax

username = *securityproperty*.**GetOriginalCreatorName( )**

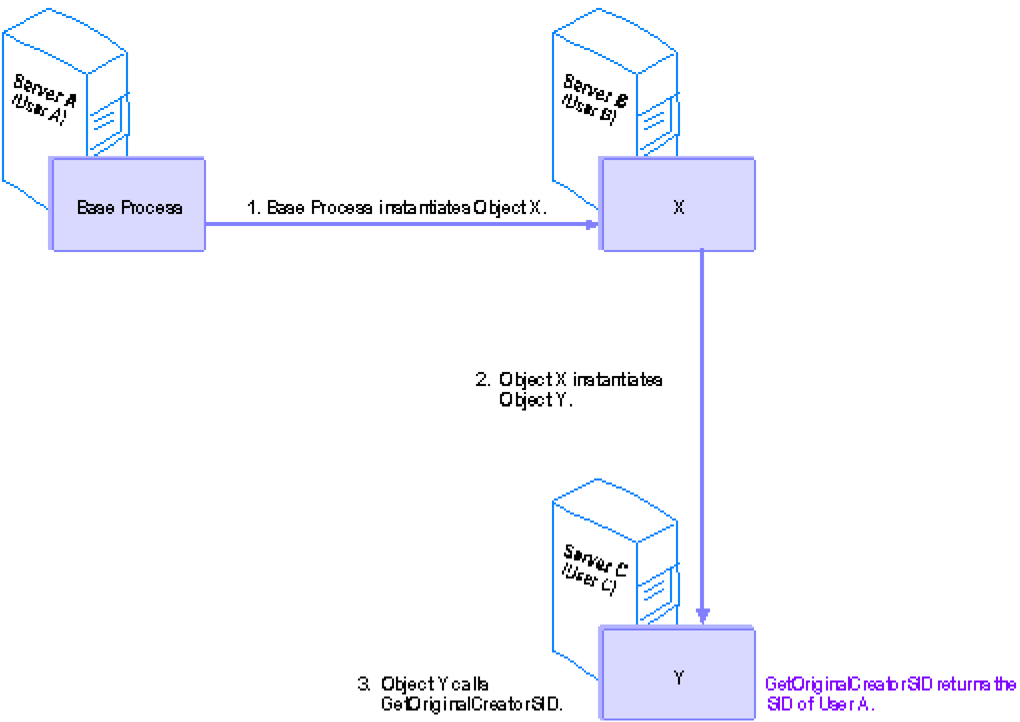
Part

username
The user name associated with the base process that initiated the activity in which the current object is executing.

securityproperty
An object variable that evaluates to a **SecurityProperty** object.

Remarks

You use the **GetOriginalCreatorName** method to determine the user name associated with the process that initiated the activity in which the current object is executing. The following scenario illustrates the functionality of the **GetOriginalCreatorName** method.



A base process running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running on server C. If object Y calls **GetOriginalCreatorName**, the name of user A is returned.

Note The path to the original creator is broken if an object is created by some other means than **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**. For example, if the base process on server A uses **CoCreateInstance** to create X, when Y calls **GetOriginalCreatorName,** the name it gets back will be the name of user B, not user A. This is because the creation sequence is traced back through the objects' context and MTS can only create a context for an object that's created with either **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

Example

See Also

Programmatic Security, Advanced Security Methods, **ObjectContext** Object

GetOriginalCreatorName Method Example

```
Public Function ComponentOriginalCreator() As String

    Dim objCtx As ObjectContext

    Set objCtx = GetObjectContext()
    ComponentOriginalCreator = _
        objCtx.Security.GetOriginalCreatorName()

End Function
```

ISecurityProperty Interface

The **ISecurityProperty** interface is used to determine the security ID of the current object's caller or creator.

Remarks

The header file for the **ISecurityProperty** interface is mtx.h. You must also link mtxguid.lib to your project to use this interface.

You obtain a reference to an object's **ISecurityProperty** interface by calling **QueryInterface** on the object's **ObjectContext**. For example:

```
m_pIObjectContext->QueryInterface (IID_ISecurityProperty,
(void**)&m_pISecurityProperty);
```

The **ISecurityProperty** interface provides the following methods.

Method	Description
GetDirectCallerSID	Retrieves the security ID of the external process that called the currently executing method.
GetDirectCreatorSID	Retrieves the security ID of the external process that directly created the current object.
GetOriginalCallerSID	Retrieves the security ID of the base process that initiated the call sequence from which the current method was called.
GetOriginalCreatorSID	Retrieves the security ID of the base process that initiated the activity in which the current object is executing.
ReleaseSID	Releases the security ID returned by one of the other **ISecurityProperty** methods.

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

ISecurityProperty::GetDirectCallerSID Method

Retrieves the security ID of the external process that called the currently executing method.

Provided By

ISecurityProperty Interface

HRESULT ISecurityProperty::GetDirectCallerSID (
  PSID* *ppSid*
);

Parameters

ppSid
  [out] A reference to the security ID of the process from which the current method was invoked.

Return Values

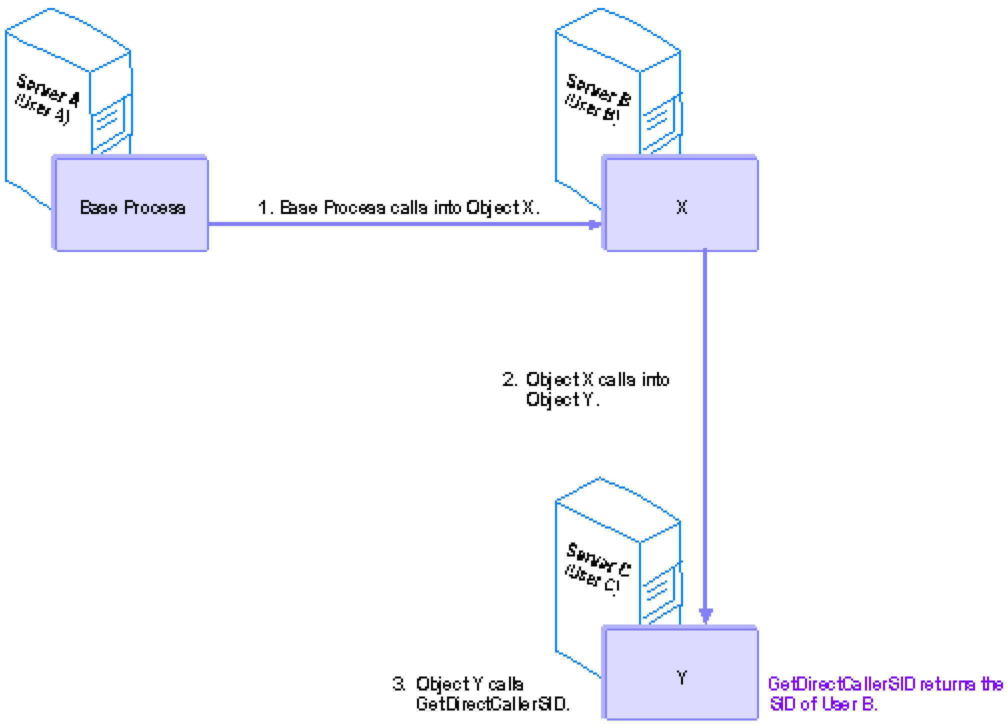
S_OK
  The security ID of the process that called the current method is returned in the parameter *ppSid*.

E_INVALIDARG
  The argument passed in the *ppSid* parameter is a NULL pointer.

E_UNEXPECTED
  An unexpected error occurred.

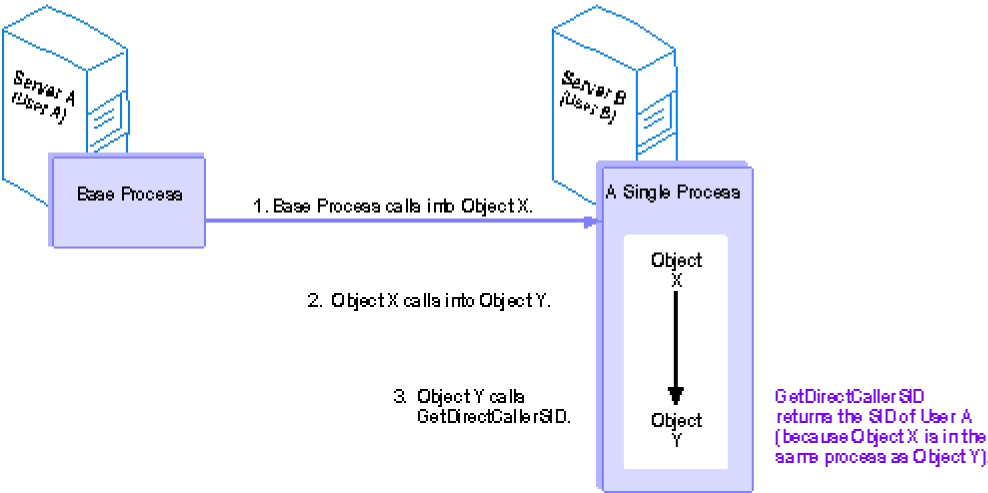
Remarks

You use the **GetDirectCallerSID** method to determine the security ID of the process that called the object's currently executing method. The following scenarios illustrate the functionality of the **GetDirectCallerSID** method.



A base process running on server A, as user A, calls into object X on server B, running as user B. Then object X calls into object Y, running on server C. If object Y calls **GetDirectCallerSID**, the the security ID of user B is returned.

Security can only be enforced across process boundaries. This means that the the security ID returned by **GetDirectCallerSID** is the the security ID associated with the process that called into the process in which the current object is running, not necessarily the immediate caller into the object itself. If an object calls into another object within the same process, when the second object calls **GetDirectCallerSID**, it will get the the security ID of the most immediate caller outside its own process boundary, not the the security ID of the object that directly called into it.



A base process, running on server A as user A, calls into object X on server B, running as user B. Then object X calls into object Y, running in the same process as object X, also on server B. When object Y calls **GetDirectCallerSID**, the the security ID of user A is returned , not the the security ID of

user B.

You must call **ReleaseSID** on a security ID when you finish using it.

Example

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

GetDirectCallerSID Method Example

```
#include <mtx.h>


IObjectContext* pIObjectContext = NULL;
ISecurityProperty* pISecurityProperty = NULL;
PSID pSid = NULL;
HRESULT hr;

// Get a reference to the ISecurityProperty interface.
pIObjectContext->QueryInterface(IID_ISecurityProperty,
   (void**)&pISecurityProperty);

// Obtain the caller's security ID.
hr = pISecurityProperty->GetDirectCallerSID(&pSid)

// Do some security checking here.

// Release the security ID.
pISecurityProperty->ReleaseSID(pSid);
```

ISecurityProperty::GetDirectCreatorSID Method

Retrieves the security ID of the current object's immediate (out-of-process) creator.

Provided By

ISecurityProperty Interface

HRESULT ISecurityProperty::GetDirectCreatorSID (
  PSID* *ppSid*
);

Parameters

ppSid
  [out] A reference to the security ID of the process that directly created the current object.

Return Values

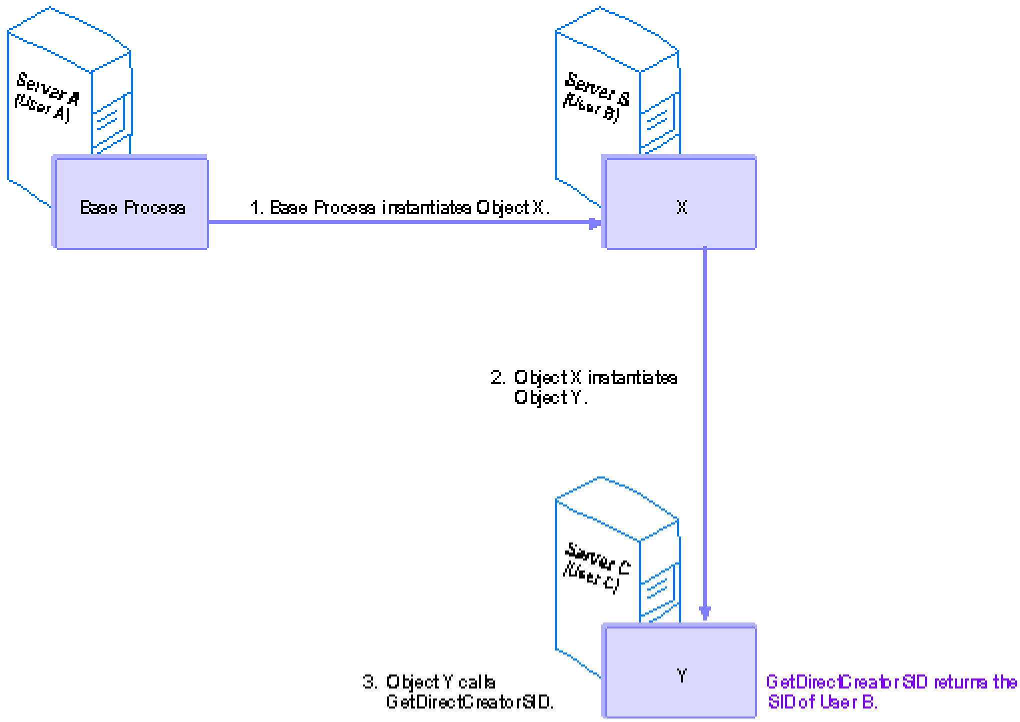
S_OK
  The security ID of the process that directly created the current object is returned in the parameter *ppSid.*

E_INVALIDARG
  The argument passed in the *ppSid* parameter is a NULL pointer.

E_FAIL
  An unexpected error occurred.

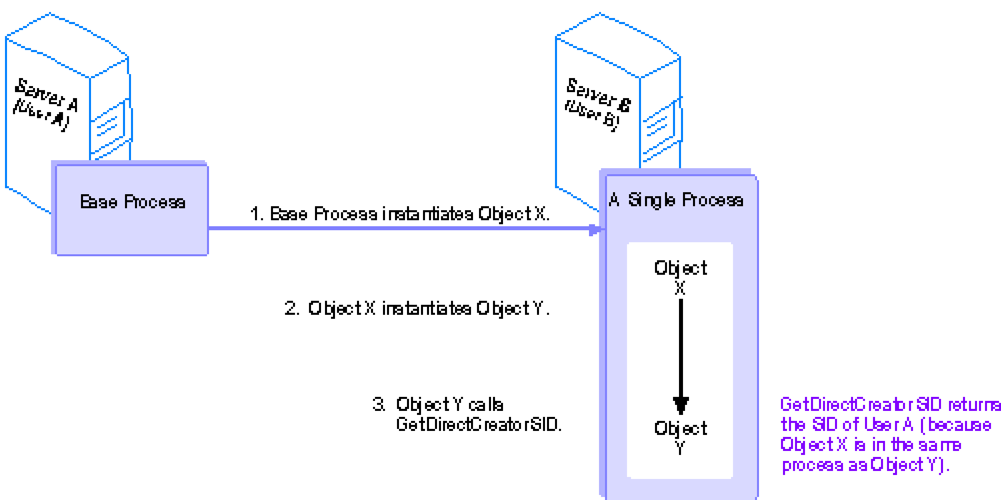
Remarks

You use the **GetDirectCreatorSID** method to determine the security ID of the process that created the current object. The following scenarios illustrate the functionality of the **GetDirectCreatorSID** method.



A base process running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running on server C. If object Y calls **GetDirectCreatorSID**, the the security ID of user B is returned.

Security can only be enforced across process boundaries. This means that if an object creates another object within the same process, when the second object calls **GetDirectCreatorSID**, it will get the the security ID of the most immediate creator outside its own process boundary, not the security ID of the object that actually created it.



A base client running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running in the same process as object X, also on server B. When object Y calls **GetDirectCreatorSID**, the the security ID of user A is returned, not the the security ID of user B.

You must call **ReleaseSID** on a security ID when you finish using it.

Example

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

GetDirectCreatorSID Method Example

```
#include <mtx.h>


IObjectContext* pIObjectContext = NULL;
ISecurityProperty* pISecurityProperty = NULL;
PSID pSid = NULL;
HRESULT hr;

// Get a reference to the ISecurityProperty interface.
pIObjectContext->QueryInterface(IID_ISecurityProperty,
   (void**)&pISecurityProperty);

// Obtain the creator's security ID.
hr = pISecurityProperty->GetDirectCreatorSID(&pSid)

// Do some security checking here.

// Release the security ID.
pISecurityProperty->ReleaseSID(pSid);
```

ISecurityProperty::GetOriginalCallerSID Method

Retrieves the security ID of the base process that initiated the sequence of calls from which the call into the current object originated.

Provided By

ISecurityProperty Interface

HRESULT ISecurityProperty::GetOriginalCallerSID (
PSID* *ppSid*
);

Parameters

ppSid
[out] A reference to the security ID of the base process that initiated the call sequence from which the current method was called.

Return Values

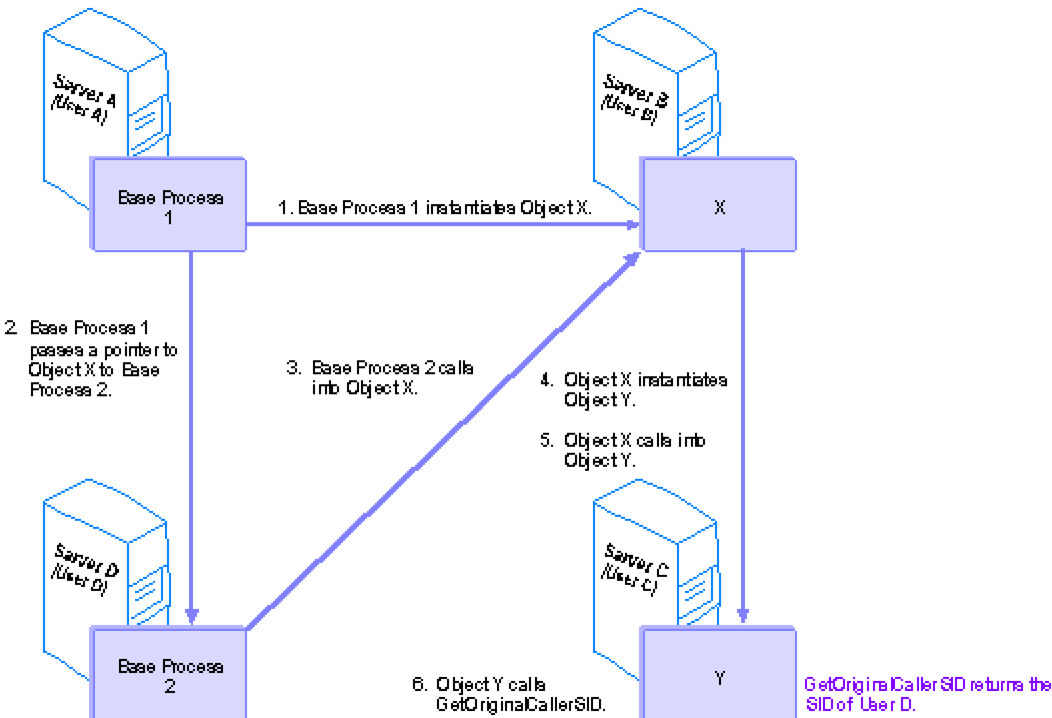
S_OK
The security ID of the base process that originated the call into the current object is returned in the parameter *ppSid*.

E_INVALIDARG
The argument passed in the *ppSid* parameter is a NULL pointer.

E_FAIL
An unexpected error occurred.

Remarks

You use the **GetOriginalCallerSID** method to determine the security ID of the original process that initiated the call sequence from which the current method was called. The following scenario illustrates the functionality of the **GetOriginalCallerSID** method.



Base process 1, running on server A as user A, creates object X on server B, running as user B. Then base process 1 passes its reference on object X to base process 2, running on server D as user D. Base process 2 uses that reference to call into object X. object X then calls into object Y, running on server C. If object Y then calls **GetOriginalCallerSID**, the the security ID of user D is returned.

Note Usually, an object's original caller is the same process as its original creator. The only situation in which the original caller and the original creator would be different is one in which the original creator passes a reference to another process, and the other process initiates the call sequence (as in the preceding example).

Note The path to the original caller is broken if any object along the chain was created by some other means than **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**. For example, if base process 1 uses **CoCreateInstance** to create X, when Y calls **GetOriginalCallerSID,** the the security ID it gets back will be the the security ID of user B, not user D. This is because the call sequence is traced back through the objects' context and MTS can only create a context for an object that's created with either **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

You must call **ReleaseSID** on a security ID when you finish using it.

Example

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

GetOriginalCallerSID Method Example

```
#include <mtx.h>


IObjectContext* pIObjectContext = NULL;
ISecurityProperty* pISecurityProperty = NULL;
PSID pSid = NULL;
HRESULT hr;

// Get a reference to the ISecurityProperty interface.
pIObjectContext->QueryInterface(IID_ISecurityProperty,
   (void**)&pISecurityProperty);

// Obtain the original caller's security ID.
hr = pISecurityProp->GetOriginalCallerSID(&pSid)

// Do some security checking here.

// Release the security ID.
pISecurityProperty->ReleaseSID(pSid);
```

ISecurityProperty::GetOriginalCreatorSID Method

Retrieves the security ID of the original base process that initiated the activity in which the current object is executing.

Provided By

ISecurityProperty Interface

HRESULT ISecurityProperty::GetOriginalCreatorSID (
PSID* *ppSid*
);

Parameters

ppSid
[out] A reference to the security ID of the base process that initiated the activity in which the current object is executing.

Return Values

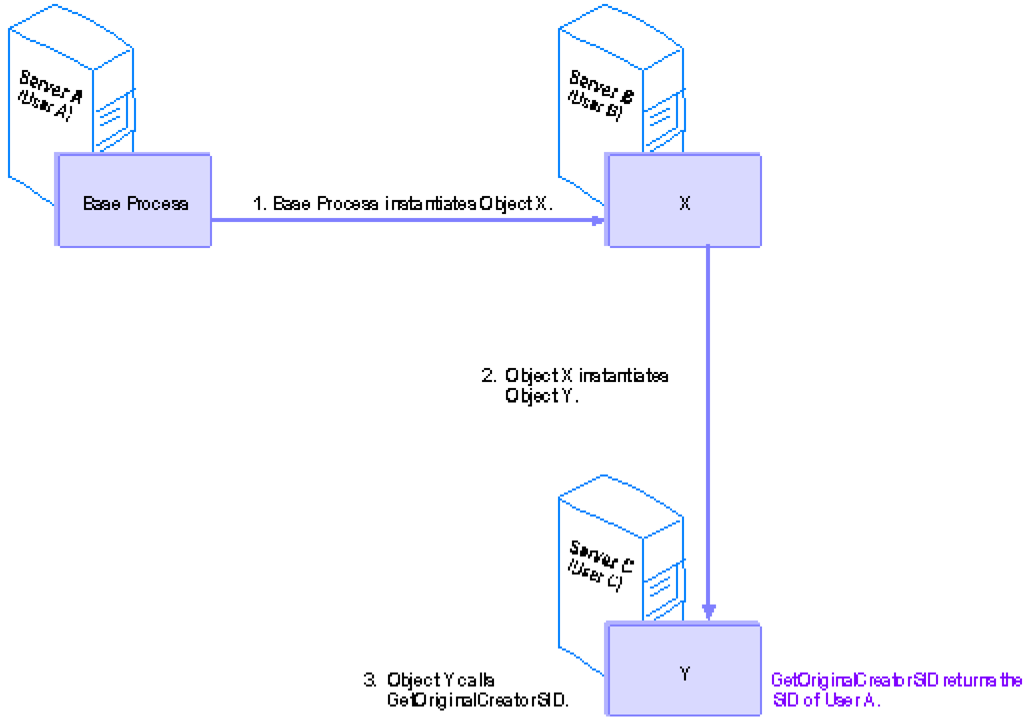
S_OK
The security ID of the original creator is returned in the parameter *ppSid*.

E_INVALIDARG
The argument passed in the *ppSid* parameter is a NULL pointer.

E_FAIL
An unexpected error occurred.

Remarks

You use the **GetOriginalCreatorSID** method to determine the security ID of the process that initiated the activity in which the current object is executing. The following scenario illustrates the functionality of the **GetOriginalCreatorSID** method.



A base process running on server A, as user A, creates object X on server B, running as user B. Then object X creates object Y, running on server C. If object Y calls **GetOriginalCreatorSID**, the the security ID of user A is returned.

Note The path to the original creator is broken if an object is created by some other means than **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**. For example, if the base process on server A uses **CoCreateInstance** to create X, when Y calls **GetOriginalCreatorSID,** the the security ID it gets back will be the the security ID of user B, not user A. This is because the creation sequence is traced back through the objects' context and MTS can only create a context for an object that's created with either **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

You must call **ReleaseSID** on a security ID when you finish using it.

Example

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

GetOriginalCreatorSID, ReleaseSID Methods Example

```
#include <mtx.h>


IObjectContext* pIObjectContext = NULL;
ISecurityProperty* pISecurityProperty = NULL;
PSID pSid = NULL;
HRESULT hr;

// Get a reference to the ISecurityProperty interface.
pIObjectContext->QueryInterface(IID_ISecurityProperty,
   (void**)&pISecurityProperty);

// Obtain the original creator's security ID.
hr = pISecurityProp->GetOriginalCreatorSID(&pSid)

// Do some security checking here.

// Release the security ID.
pISecurityProperty->ReleaseSID(pSid);
```

ISecurityProperty::ReleaseSID Method

Releases a security ID that was obtained from the **GetDirectCallerSID**, **GetDirectCreatorSID**, **GetOriginalCallerSID**, or **GetOriginalCreatorSID** method.

Provided By

ISecurityProperty Interface

HRESULT ISecurityProperty::ReleaseSID (
PSID *pSid*
);

Parameters

pSid
[in] A reference to a security ID that was obtained by invoking one of the **ISecurityProperty** methods.

Return Values

S_OK
The security ID, passed in the *pSid* parameter, was released.

E_INVALIDARG
The argument passed in the *pSid* parameter is not a reference to a security ID.

Remarks

You should always invoke the **ReleaseSID** method to release any security ID pointers returned by the **GetDirectCallerSID, GetDirectCreatorSID, GetOriginalCallerSID,** and **GetOriginalCreatorSID** methods of the **ISecurityProperty** interface.

See Also

Programmatic Security, Advanced Security Methods, **IObjectContext** Interface

MTS Error Codes

The following errors can be returned by Microsoft Transaction Server (MTS) objects.

S_OK
The call succeeded.

E_INVALIDARG
One or more of the arguments passed in is invalid.

E_UNEXPECTED
An unexpected error occurred.

CONTEXT_E_NOCONTEXT
The current object doesn't have a context associated with it. This is probably either because its component hasn't been installed in a package or it wasn't created with one of the MTS **CreateInstance** methods.

CONTEXT_E_ROLENOTFOUND
The role specified in the *szRole* parameter in the **IObjectContext::IsCallerInRole** method does not exist.

E_OUTOFMEMORY
There's not enough memory available to instantiate the object. This error code can be returned by **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

REGDB_E_CLASSNOTREG
The specified component is not registered as a COM component. This error code can be returned by **IObjectContext::CreateInstance** or **ITransactionContext::CreateInstance**.

DISP_E_ARRAYISLOCKED
One or more of the arguments passed in contains an array that is locked. This error code can be returned by the **ISharedProperty::put_Value** method.

DISP_E_BADVARTYPE
One or more of the arguments passed in isn't a valid VARIANT type. This error code can be returned by the **ISharedProperty::put_Value** method.

MTS Supported Variant Types

The following Automation types are supported by Microsoft Transaction Server.

VT_BOOL	**VT_LPSTR**
VT_BSTR	**VT_LPWSTR**
VT_CARRAY	**VT_NULL**
VT_CLSID	**VT_PTR**
VT_CY	**VT_R4**
VT_DATE	**VT_R8**
VT_DECIMAL	**VT_SAFEARRAY**
VT_DISPATCH	**VT_UINT**
VT_EMPTY	**VT_UI1**
VT_ERROR	**VT_UI2**
VT_HRESULT	**VT_UI4**
VT_INT	**VT_UI8**
VT_I1	**VT_UNKNOWN**
VT_I2	**VT_USERDEFINED**
VT_I4	**VT_VARIANT**
VT_I8	**VT_VOID**

The following types are not supported and will cause the server process to terminate with an error.

VT_FILETIME	**VT_BLOB**
VT_STREAM	**VT_STORAGE**
VT_STREAMED_OBJECT	**VT_STORED_OBJECT**
VT_BLOB_OBJECT	**VT_CF**

MTS Administrative Reference

The Microsoft Transaction Server (MTS) Administrative reference provides object and method information for Microsoft® Visual Basic or Microsoft Visual Basic Scripting Edition (VBScript) programmers, and interface and function information for Microsoft Visual C++® programmers. In addition, this technical reference provides a detailed description of the collections used by the administration objects.

This section contains the following sections:

Using MTS Administration Objects

Using MTS Collection Types

Automating MTS Administration With Visual Basic

MTS Administration Object Methods

Automating MTS Administration With Visual C++

See Also

Automating MTS Administration

Automating MTS Administration With Visual Basic

This reference topic provides guidance for Microsoft® Visual Basic or Microsoft Visual Basic Scripting Edition (VBScript) programmers who want to use the administration objects to automate tasks that an administrator performs using the Microsoft Transaction Server (MTS) Explorer. The MTS Visual Basic reference contains the following topics:

MTS Visual Basic Error Codes
MTS Administration Object Methods

See Also

Automating MTS Administration

Using MTS Administration Objects

Use the MTS administration objects to automate administration for MTS packages.

This section describes how the following administration objects are used:

MTS Catalog Object

MTS CatalogObject Object

MTS CatalogCollection Object

MTS PackageUtil Object

MTS ComponentUtil Object

MTS RemoteComponentUtil Object

MTS RoleAssociationUtil Object

MTS Catalog Object

The Catalog object enables you to connect to an MTS catalog and access collections. This general administration object supports the following methods.

Method	Description
GetCollection	Gets a collection on the catalog without reading any objects from the catalog.
Connect	Connects to a remote catalog and returns a root collection.
MajorVersion	Returns the major version number of the catalog.
MinorVersion	Returns the minor version number of the catalog.

See Also

MTS CatalogObject Object, **MTS CatalogCollection Object**, **MTS PackageUtil Object**, **MTS ComponentUtil Object**, **MTS RemoteComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS CatalogObject Object

The **CatalogObject** object allows you to get and set object properties. This general administration interface supports the following methods.

Method	Description
Value	Gets and sets a property value
Key	Gets the value of the key property
Name	Gets the name of the object
IsPropertyReadOnly	**True** if the property cannot be set
IsPropertyWriteOnly	**True** if the property only supports set
Valid	**True** if all properties were successfully read from the catalog data store

See Also

MTS Catalog Object, **MTS CatalogCollection Object**, **MTS PackageUtil Object**, **MTS ComponentUtil Object**, **MTS RemoteComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS CatalogCollection Object

Use the **CatalogCollection** object to enumerate, add, delete, and modify catalog objects and to access related collections This general administration interface supports the following methods.

Method	Description
Item	Returns an object by index. The index is zero-based.
Count	Returns number of objects in the collection.
Remove	Removes an item according to its index position.
Add	Adds an object to the collection.
Populate	Reads all the collection objects from the catalog data store.
SaveChanges	Saves changes made to the collection into the catalog data store.
GetCollection	Gets a collection related to a specific object.
Name	Gets the name of a collection.
AddEnabled	Returns true if the **Add** method is enabled.
RemoveEnabled	Returns true if the **Remove** method is enabled.
GetUtilInterface	Gets the utility interface for the collection.
PopulateByKey	Reads the selected collection objects from the catalog data store.
DataStoreMajorVersion	Returns the major version number of the catalog data store.
DataStoreMinorVersion	Returns the minor version number of the catalog data store.

See Also

Add Method (CatalogCollection), **Remove Method (CatalogCollection)**, **MTS Catalog Object**, **MTS CatalogObject Object**, **MTS PackageUtil Object**, **MTS ComponentUtil Object**, **MTS RemoteComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS PackageUtil Object

The **PackageUtil** object enables installing and exporting a package. Instantiate this object by calling **GetUtilInterface** on a **Packages** collection.

This utility administration interface supports the following methods:

Method	Description
InstallPackage	Installs a pre-built package.
ExportPackage	Exports a package.
ShutdownPackage	Shuts down a package, thereby terminating that server process.

See Also

MTS Catalog Object, **MTS CatalogObject Object**, **MTS CatalogCollection Object**, **MTS ComponentUtil Object**, **MTS RemoteComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS ComponentUtil Object

Call the **ComponentUtil** object to install a component in a specific collection and import components registered as in-process servers. Create this object by calling **GetUtilInterface** on a **ComponentsInPackage** collection. This utility administrationinterface supports the following methods.

Method	Description
InstallComponent	Installs a component from a DLL.
ImportComponent	Imports a component that is already registered as an in-process server. Supply the CLSID of the component.
ImportComponentByName	Imports a component that is already registered as an in-process server. Supply the ProgID of the component.
GetCLSIDS	Returns an array of installable CLSIDs in the DLL/type library. Note that changes are not made to the data store.

See Also

MTS Catalog Object, **MTS CatalogObject Object**, **MTS CatalogCollection Object**, **MTS PackageUtil Object**, **MTS RemoteComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS RemoteComponentUtil Object

Using the **RemoteComponentUtil** object, you can program your application to pull remote components from a package on a remote server. Instantiate this object by calling **GetUtilInterface** on a **RemoteComponents** collection. This utility administration interface supports the following methods.

Method	Description
InstallRemoteComponent	Pulls remote components from a package on a remote server. Supply the package ID and CLSID.
InstallRemoteComponentByName	Pulls remote components from a package on a remote server. Supply the package name and ProgID.

See Also

GetUtilInterface Method (CatalogCollection), **MTS Catalog Object**, **MTS CatalogObject Object**, **MTS CatalogCollection Object**, **MTS PackageUtil Object**, **MTS ComponentUtil Object**, **MTS RoleAssociationUtil Object**

MTS RoleAssociationUtil Object

Call methods on the **RoleAssociationUtil** object to associate roles with a component or interface. Create this object by calling the **GetUtilInterface** method on a **RolesForPackageComponent** or **RolesForPackageComponentInterface** collection. This utility administration interface supports the following methods.

Method	Description
AssociateRole	Associates the role by role ID with the component or interface.
AssociateRoleByName	Associates the role by role name with the component or interface.

See Also

GetUtilInterface Method (CatalogCollection), **MTS Catalog Object**, **MTS CatalogObject Object**, **MTS CatalogCollection Object**, **MTS PackageUtil Object**, **MTS ComponentUtil Object**, **MTS RemoteComponentUtil Object**

Using MTS Collection Types

This topic describes the collections and collection properties supported by the MTS catalog. Additional collections and properties may be added in future versions. Use the **DataStoreMajorVersion** and **DataStoreMinorVersion** properties of a collection to distinguish between catalog versions.

The MTS catalog data store supports the following collection types:

MTS LocalComputer Collection
MTS ComputerList Collection
MTS Packages Collection
MTS ComponentsInPackage Collection
MTS RemoteComponents Collection
MTS InterfacesForComponent and InterfacesForRemoteComponent Collections
MTS RolesForPackageComponent and RolesForPackageComponentInterface Collections
MTS MethodsForInterface Collection
MTS RolesInPackage Collection
MTS UsersInRole Collection
MTS ErrorInfo Collection
MTS PropertyInfo Collection
MTS RelatedCollectionInfo Collection

MTS LocalComputer Collection

The **LocalComputer** contains a single object that corresponds to the computer whose catalog that you are accessing. If you call the **Connect** method on the **Catalog** object, the **LocalComputer** collection contains information about the computer whose catalog you are accessing. This collection does not support the **Add, Remove** or **GetUtilInterface** methods.

The following table provides a list of the properties supported by the **LocalComputer** collection.

Property	Description
Name	"My Computer". Data Type: String Default value: N/A Access: Read only.
Description	Description of the computer. Data Type: String Default value: Empty string Access: Read/write
Transaction Timeout	The transaction timeout setting in seconds. Data Type: Integer Default value: 60 Access: Read/write
RemoteComponent InstallPath	The path in which remote component files will be installed when remote components are configured Data Type: String Default value: the subdirectory "Remote" in the MTS install path Access: Read/write
PackageInstall Path	The default path in which component files will be installed when pre-built packages are installed. Data Type: String Default value: the subdirectory "Packages" in the MTS install path Access: Read/write
ResourcePooling Enabled	"Y" enables resource pooling. "N" disables resource pooling. Note that resource pooling is currently only supported by the ODBC resource dispenser. Data Type: String Default value: "Y" Access: Read/write
ReplicationShare	When replicating the MTS catalog, the name of a share that the target system should use to access installed packages. Data Type: String Default value: Empty string Access: Read/write
RemoteServer Name	The computer name to be generated when you use the MTS Explorer to create a client executable. If this string is blank or empty, the physical computer name of the exporting computer is used. If you put the name of the remote server as the string, the application executable generated by the MTS Explorer points to that remote server name. Data Type: String Default value: Empty string Access: Read/write

See Also

RelatedCollectionInfo, **PropertyInfo**, **ErrorInfo**

MTS ComputerList Collection

The **ComputerList** collection provides access to the list of computers shown in the MTS Explorer Computers folder. This collection supports the **Add** and **Remove** methods. This collection does not support the **GetUtilInterface** method.

The following table provides a list of the properties supported by the **ComputerList** collection.

Property	Description
Name	The name of the computer. Data Type: String Default value: "NewComputer" Access: Read/Write while using the **Add** method. After adding, read-only.

See Also

RelatedCollectionInfo, **PropertyInfo**, **ErrorInfo**, **Add Method (CatalogCollection)**, **Remove Method (CatalogCollection)**

MTS Packages Collection

As the top-level collection managed by the MTS Explorer, the **Packages** collection contains the packages installed on the local machine running MTS. Packages contain a set of components that run in the same server process, and define declarative security constructs that determine access to components at run time. The **Packages** collection supports the **Add** method and **Remove** method on the **CatalogCollection** object. In addition, the **GetUtilInterface** method of this collection returns a **PackageUtil** object which can be used to install and export packages.

The following table provides a list of the properties supported by the **CatalogObject** objects within the **Packages** collection.

Property	Description
Name	Name of the package. Data Type: String Default value: "New package" Access: Read/Write
ID	A universally unique identifier (UUID) for the package. Data Type: String Default value: A unique identifier is generated Access: Read/Write when using the **Add** method. Read-only after using the **Add** method.
Description	Describes the package. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read/Write
IsSystem	Identifies an MTS system package. "N" signifies that the package is not a Transaction Server system package, and "Y" indicates that package is an MTS system package. Data Type: String Default value: "N" Access: Read only
Authentication	Sets authentication level for calls. Possible values are 0 through 6, which correspond to the Remote Procedure Call (RPC) authentication settings. Data Type: Long Default value: 4 Access: Read/Write
ShutdownAfter	Sets the delay before shutting down a server process after it becomes idle. Shutdown latency ranges from 0 to 1440 minutes. Data Type: Long Default value: 3 Access: Read/Write
RunForever	Enables a server process to continue if a package is idle. If value is set to "Y", the server process will not shut down when left idle. If set to "N", the process will shut down according the value set by the **ShutDownAfter** property. Data Type: String Default value: "N" Access: Read/Write
SecurityEnabled	Checks the security credentials of any client that calls the package if value is set to "Y." Data Type: String

	Default value: “N” Access: Read/Write
Identity	Sets the server process identity for the package. Specify a valid Windows NT user account or "Interactive User" to have the package assume the identity of the current logged-on user. Data Type: String Default value: “Interactive User” Access: Read/Write
Password	Sets the password used by the server process to log on under the identity above. Data Type: String Default value: None Access: Write only
Activation	Sets the package level activation property to either “Local” or “Inproc”. The Local setting determines that objects within the package will run within a dedicated local server process. A package running under the Local activation setting is a “server package”. The Inproc activation setting means objects run in their creator’s process. A package running under the Inproc activation setting is a “library package” Data Type: String Default Value: “Local” Access: Read/Write
Changeable	Sets whether changes to the package settings, or those of its components, are allowed (either programmatically, or through the MTS UI). Data Type: String Default Value: Y Access: Read/Write
Deleteable	Sets whether the package or its components can be deleted (either programmatically, or through the MTS UI). Data Type: String Default Value: “Y” Access: Read/Write
CreatedBy	Informational string to describe the package creator. Data Type: String Default Value: Empty string Access: Read/Write

See Also

ComponentsInPackage, **RolesInPackage**, **RelatedCollectionInfo**, **PropertyInfo**, **ErrorInfo**

MTS ComponentsInPackage Collection

The **ComponentsInPackage** collection contains the set of components that run in the same server process and compose a package. This collection supports the **Remove** method. The **Add** method is not supported. You must use the **ComponentUtil** object to install components into the package.

The following table provides a list of the properties supported by the **CatalogObjects** within the **ComponentsInPackage** collection.

Property	Description
ProgID	The name that identifies the component. Data Type: String Default value: None Access: Read only
CLSID	The universally unique identifier (UUID) for the component. Data Type: String Default value: None Access: Read only
Transaction	Determines how a component supports transactions. Must be one of the following transaction settings: "Required" "Requires New" "Not Supported" "Supported" Data Type: String Default value: "Not supported" Access: Read/Write
Description	Describes the component. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read/Write
PackageID	Defines the identity of the owning package. Data Type: String Default value: None Access: Read only
PackageName	Defines the name of the owning package. Data Type: String Default value: "New Package" Access: Read only
ThreadingModel	Determines how instances of the component are assigned to threads for method execution. Possible values are those supported by Component Object Model (COM). Data Type: String Default value: None Access: Read only
SecurityEnabled	Checks the security credentials of any client that calls the component if value is set to "Y." Data Type: String Default value: "Y" Access: Read/Write
DLL	Displays the name of the DLL containing the component implementation. Data Type: String Default value: None

	Access: Read only
IsSystem	Identifies an MTS system component. “N” signifies that the package is not a Transaction Server system component, and “Y” indicates that package is an MTS system component. Data Type: String Default value: “N” Access: Read only

See Also

InterfacesForComponent, **RolesForPackageComponent**, **RelatedCollectionInfo**, **PropertyInfo**, **ErrorInfo**

MTS RemoteComponents Collection

Use the Microsoft Transaction Server (MTS) Explorer on a client computer to add remote component entries that see components installed on a remote server. This is often described as "pulling" remote component information from a server. Configuring remote components automatically copies proxy/stub DLLs and type libraries from the server to the client. The **RemoteComponents** collection supports the **Remove** method. This collection does not support the **Add** method on the **CatalogCollection** object. Instead you must call **GetUtilInterface** to obtain the **RemoteComponentUtil** object in order to add new remote components.

The following table provides a list of the properties supported by the **CatalogObjects** within the **RemoteComponents** collection.

Property	Description
ProgID	The name that identifies the remote component. Data Type: String Default value: None Access: Read only
CLSID	The universally unique identifier (UUID) for the component. Data Type: String Default value: None Access: Read only
Description	Describes the remote component. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read/Write
Server	Name of the server hosting the remote component. Data Type: String Default value: None Access: Read only

See Also

InterfacesForRemoteComponent, **RelatedCollectionInfo**, **PropertyInfo**, **ErrorInfo**

MTS InterfacesForComponent and InterfacesForRemoteComponent Collections

The **InterfacesForComponent** and **InterfacesForRemoteComponent** collections provide information about a selected interface. **InterfacesForComponent** or **InterfacesForRemoteComponent** collections are listed for each component, and can be used by an administrator to identify or manage the interface. These collections do not support the **Add**, **Remove**, or **GetUtilInterface** methods.

The following table provides a list of the properties supported by the **CatalogObjects** within the **InterfacesForComponent** and the **InterfacesForRemoteComponent** collections.

Property	Description
Name	Displays the friendly name of the interface. Data Type: String Default value: None Access: Read only
ID	Displays the unique interface identifier (IID). Data Type: String Default value: None Access: Read only
Description	Describes the interface. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read/Write
ProxyCLSID	Displays the CLSID of the proxy/stub. Data Type: String Default value: None Access: Read only
ProxyDLL	Displays the file name of the proxy/stub DLL. Data Type: String Default value: None Access: Read only
ProxyThreadingModel	Displays the threading model of the selected proxy/stub. Data Type: String Default value: None Access: Read only
TypeLibID	Displays the UUID of the type library. Data Type: String Default value: None Access: Read only
TypeLibVersion	Displays the version of the type library. Data Type: String Default value: None Access: Read only
TypeLibLangID	Displays the language identification number of the type library. Data Type: String Default value: None Access: Read only
TypeLibPlatform	Displays the platform of the type library. Data Type: String

	Default value: None Access: Read only
TypeLibFile	Displays the file name of the type library. Data Type: String Default value: None Access: Read only

See Also

MethodsForInterface (InterfacesForComponent only), **RolesForPackageComponentInterface (InterfacesForComponent only)**, **RelatedCollectionInfo**, **PropertyInfo**, **ErrorInfo**

MTS RolesForPackageComponent and RolesForPackageComponentInterface Collections

The **RoleForPackageComponent** and **RolesForPackageComponentInterface** collections contain the roles associated with a component or interface. You add existing roles to these collections from a package's **RolesInPackage** collection. The **Add** method is not supported by this collection. Use the RoleAssociationUtil methods to add roles to this collection. The CatalogCollection **Remove** method is supported by this collection.

The following table provides a list of the properties supported by the **CatalogObject(s)** within the **RolesForPackageComponent** and the **RolesForPackageComponentInterface** collections.

Property	Description
Name	Displays the name of the role associated with a component. Data Type: String Default value: None Access: Read only
ID	Displays the universally unique identifier (UUID) of the role. Data Type: String Default value: None Access: Read only
Description	Describes the role. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read only

See Also

RelatedCollectionInfo, **PropertyInfo**, **ErrorInfo**

MTS MethodsForInterface Collection

The **MethodsForInterface** collection contains the methods defined in an interface. Method properties are used to display information about the methods exposed by an interface. This collection does not support the **Add**, **Remove**, or **GetUtilInterface** methods.

The following table provides a list of the properties supported by the **CatalogObjects** within the **MethodsForInterface** collection.

Property	Description
Name	Displays the name of a method. Data Type: String Default value: None Access: Read only
Description	Describes the method. Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read only

See Also

RelatedCollectionInfo, **PropertyInfo**, **ErrorInfo**

MTS RolesInPackage Collection

The **RolesInPackage** collection defines a class of users for a set of components in a package. Each role defines a set of users allowed to invoke interfaces on a component. Roles can be applied to both components and component interfaces. This collection supports the **Add** and **Remove** methods. This collection does not support the **GetUtilInterface** method.

The following table provides a list of the properties supported by the **CatalogObjects** within the **RolesInPackage** collection.

Property	Description
Name	The role name. Data Type: String Default value: “New Role” Access: Read/write
ID	The universally unique identifier (UUID) for the role. Data Type: String Default value: A unique identifier is generated. Access: Read/Write while using the Add method. After adding an object, Read-only.
Description	Describes the new **Role.** Description fields hold a maximum of 500 characters. Data Type: String Default value: None Access: Read/write

See Also

UsersInRole, **RelatedCollectionInfo**, **PropertyInfo**, **ErrorInfo**

MTS UsersInRole Collection

The **UsersInRole** collection lists the members of the class of users that have been authorized to invoke methods on the component or component interface associated with the role. This collection supports the **Add** and **Remove** methods. This collection does not support the **GetUtilInterface** method.

The following table provides a list of the properties supported by the **CatalogObjects** within the **UsersInRole** collection.

Property	Description
User	The name of an NT user account or group. Data Type: String Default value: “New User” Access: Read/Write while using the Add method. After adding, read-only.

See Also

RelatedCollectionInfo, **PropertyInfo**, **ErrorInfo**

MTS ErrorInfo Collection

The **ErrorInfo** collection is used to retrieve extended error information about methods that deal with multiple objects. This collection does not support the **Add**, **Remove**, or **GetUtilInterface** methods. Use the **GetCollection** method on a collection to access the **ErrorInfo** collection associated with the original collection. The **ErrorInfo** collection is accessible from any collection except **ErrorInfo**, **RelatedCollectionInfo**, and **PropertyInfo**. When calling methods on a utility object, extended error information may be created in the **ErrorInfo** collection associated with the collection used to create the utility object.

The following table provides a list of the properties supported by the **CatalogObjects** within the **ErrorInfo** collection.

Property	Description
Name	Name of the object or file. Data Type: String Default value: None Access: Read only.
ErrorCode	Error code for the object or file. Data Type: Long Default value: None Access: Read only

See Also

MTS LocalComputer Collection, **MTS ComputerList Collection**, **MTS Packages Collection**, **MTS ComponentsInPackage Collection**, **MTS RemoteComponents Collection**, **MTS InterfacesForComponent and InterfacesForRemoteComponent Collections**, **MTS RolesForPackageComponent and RolesForPackageComponentInterface Collections**, **MTS MethodsForInterface Collection**, **MTS RolesInPackage Collection**, **MTS UsersInRole Collection**, **MTS PropertyInfo Collection**, **MTS RelatedCollectionInfo Collection**

MTS PropertyInfo Collection

The **PropertyInfo** collection is used to retrieve information about the properties that a specified collection supports. This collection does not support the **Add**, **Remove**, or **GetUtilInterface** methods. The **PropertyInfo** collection is accessible from any collection by using the **GetCollection** method.

The following table provides a list of the properties supported by the **CatalogObject(s)** within the **PropertyInfo** collection.

Property	Description
Name	Name of the property. Data Type: String Default value: None Access: Read only

See Also

MTS LocalComputer Collection, **MTS ComputerList Collection**, **MTS Packages Collection**, **MTS ComponentsInPackage Collection**, **MTS RemoteComponents Collection**, **MTS InterfacesForComponent and InterfacesForRemoteComponent Collections**, **MTS RolesForPackageComponent and RolesForPackageComponentInterface Collections**, **MTS MethodsForInterface Collection**, **MTS RolesInPackage Collection**, **MTS UsersInRole Collection**, **MTS ErrorInfo Collection**, **MTS RelatedCollectionInfo Collection**

MTS RelatedCollectionInfo Collection

The **RelatedCollectionInfo** collection is used to retrieve information about other collections related to the collection from which this collection is called. The **RelatedCollectionInfo** collection is accessible from any collection by using the **GetCollection** method. The **RelatedCollectionInfo** collection will contain one object for each collection that is accessible from the original collection. Related collections follow the MTS Explorer folder hierarchy. This collection does not support the **Add**, **Remove**, or **GetUtilInterface** methods.

The following table provides a list of the properties supported by the **CatalogObjects** within the **RelatedCollectionInfo** collection.

Property	Description
Name	Name of the related collection. Data Type: String Default value: None Access: Read only

See Also

MTS LocalComputer Collection, **MTS ComputerList Collection**, **MTS Packages Collection**, **MTS ComponentsInPackage Collection**, **MTS RemoteComponents Collection**, **MTS InterfacesForComponent and InterfacesForRemoteComponent Collections**, **MTS RolesForPackageComponent and RolesForPackageComponentInterface Collections**, **MTS MethodsForInterface Collection**, **MTS RolesInPackage Collection**, **MTS UsersInRole Collection**, **MTS ErrorInfo Collection**, **MTS PropertyInfo Collection**

MTS Visual Basic Error Codes

The following table lists the error codes returned by methods called on the MTS catalog collection and catalog utility objects.

Error code	Description
Visual Basic run-time error 5	Indicates one of the following: An invalid collection or property name was entered. An out parameter was NULL. The value is not one of the supported values or falls outside the supported range. The property is read-only. The property cannot be changed after the object is created. An invalid index was entered.
Visual Basic run-time error 445	Object has been removed from the collection or the method is not supported on this object.
mtsErrObjectErrors	Errors were encountered processing some objects or file. See the **ErrorInfo** collection for object and file-specific error codes.
mtsErrNoUser	User ID for user in role is not valid.
mtsErrUserPasswdNotValid	Package identity user ID and/or password are not valid.
mtsErrAuthenicationLevel	Required authentication level (package privacy) could not be set for package updates.
mtsErrPDFReadFail	An error occurred reading the package file.
mtsErrPDFVersion	Package file version is invalid.
mtsErrBadPath	Package file path is invalid.
mtsErrPackageExists	Package with the same ID is already installed.
mtsErrRoleExists	A role with the same ID is already installed. The role ID in the package file is likely corrupted.
mtsErrCantCopyFile	Errors occurred copying one or more files to the install directory.
mtsErrInvalidUserids	One or more user IDS for roles were invalid.
mtsErrCLSIDOrIIDMismatch	One or more component or interface identifiers in a component DLL do not match the identifiers saved in the package file. The package file is out of date.
mtsErrPackDirNotFound	Package install directory is invalid due to general registry read/write errors.
mtsErrPDFWriteFail	An error occurred writing the package file.
mtsErrNoTypeLib	Could not find the type library for one or more components.

The following table lists the object or file-specific error codes returned in **ErrorInfo** collections.

Error code	Description
mtsErrObjectInvalid	One or more object properties is corrupted or invalid.
mtsErrKeyMissing	One or more objects is not found in the catalog data store.
mtsErrAlreadyInstalled	Component is already installed.
mtsErrDownloadFailed	One or more component files could not be copied to the client.
mtsErrRemoteInterface	No interface information is available for the component. Component files could not be downloaded.
mtsErrCoReqCompInstalled	Component in the same DLL file is already installed.
mtsErrNoRegistryCLSID	Component's CLSID is corrupted.
mtsErrBadRegistryProgID	Component's ProgID is corrupted.
mtsErrDllLoadFailed	Component's DLL could not be loaded.
mtsErrDllRegisterServer	**DllRegisterServer** method failed during component self-registration.
mtsErrNoServerShare	No file share is available on the server to copy component files from the network path.
mtsErrNoAccessToUNC	Network path registered for this component could not be accessed.
mtsErrBadRegistryLibID	Component type library ID is corrupted.
mtsErrTreatAs	Component **TreatAs** key was found, but is not supported.
mtsErrBadForward	IID forward entry is corrupted.
mtsErrBadIID	IID is corrupted.
mtsErrRegistrarFailed	Component registrar method failed during component install.
mtsErrCompFileDoesNotExist	Component file does not exist.
mtsErrCompFileLoadDLLFail	DLL file could not be loaded.
mtsErrCompFileGetClassObj	**DllGetClassObject** function call failed during the DLL self-registration process.
mtsErrCompFileClassNotAvail	Class coded in the type library was not supported.
mtsErrCompFileBadTLB	Type library was corrupted.
mtsErrCompFileNotInstallable	File does not contain COM components or type library information.
mtsErrNotChangeable	Changes to this object and sub-objects have been disabled.

mtsErrNotDeletable	Delete function for this object has been disabled.
mtsErrSession	Catalog version is not supported.

The following tables lists general read and write registry errors.

Error Code	Description
mtsErrNoRegistryRead	Access control failure reading a registry key.
mtsErrNoRegistryWrite	Access control failure writing a registry key.
mtsErrNoRegistryRepair	Access control failure writing a registry key.

MTS Administration Object Methods

The following topics list the methods of the MTS administration objects:

Add Method (CatalogCollection)
AddEnabled Property (CatalogCollection)
AssociateRole Method (RoleAssociationUtil)
AssociateRoleByName Method (RoleAssociationUtil)
Connect Method (Catalog)
Count Property (CatalogCollection)
DataStoreMajorVersion Property (CatalogCollection)
DataStoreMinorVersion Property (CatalogCollection)
ExportPackage Method (PackageUtil)
GetCLSIDs Method (ComponentUtil)
GetCollection Method (Catalog)
GetCollection Method (CatalogCollection)
GetUtilInterface Method (CatalogCollection)
ImportComponent Method (ComponentUtil)
ImportComponentByName Method (ComponentUtil)
InstallComponent Method (ComponentUtil)
InstallPackage Method (PackageUtil)
InstallRemoteComponent Method (RemoteComponentUtil)
InstallRemoteComponentByName Method (RemoteComponentUtil)
IsPropertyReadOnly Property (CatalogObject)
IsPropertyWriteOnly Property (CatalogObject)
Item Property (CatalogCollection)
Key Property (CatalogObject)
MajorVersion Property (Catalog)
MinorVersion Property (Catalog)
Name Property (CatalogObject)
Name Property (CatalogCollection)
Populate Method (CatalogCollection)
PopulateByKey Method (CatalogCollection)
Remove Method (CatalogCollection)
RemoveEnabled Property (CatalogCollection)
SaveChanges Method (CatalogCollection)
ShutdownPackage Method
Valid Property (CatalogObject)
Value Property (CatalogObject)

See the Automating MTS Administration topic for sample code that demonstrates how these methods are used to program administration using Microsoft® Visual Basic or Microsoft Visual Basic Scripting Edition (VBScript).

Add Method (CatalogCollection)

Adds a member to a collection object and returns the **CatalogObject** object.

Syntax

object**.Add**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

Call the **Add** method to create a new object in a collection. This method is supported in the following collections:

Packages

RolesInPackage

UsersInRole

To install or create objects in other collections, use the catalog utility interfaces. Note that you must call the **SaveChanges** method to write the new object to the catalog data store.

For a list of the MTS collections and their properties, see Using MTS Collections.

See Also

MTS Packages Collection, **MTS RolesInPackage Collection**, **MTS UsersInRole Collection**, **AddEnabled Property (CatalogCollection)**

AddEnabled Property (CatalogCollection)

Returns a **Boolean** value indicating whether the **Add** method is enabled on this collection.

Syntax

object.**AddEnabled**

Parameters

object
   Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

If the value returned is **True**, then you can call the **Add** method to create a new object in a collection. If the value returned is **False**, you must use the catalog utility object methods to create a new object.

For a list of the MTS collections and their properties, see the Using MTS Collections topic.

See Also

Add Method (CatalogCollection)

AssociateRole Method (RoleAssociationUtil)

Associates a role with a component or component interface.

Syntax

object.**AssociateRole(***ID***)**

Parameters

object
    Required. An object variable that evaluates to a **RoleAssociationUtil** object.
ID
    Required. A **String** expression that specifies the role ID of the roles to associate with the object.

Remarks

The changes are applied immediately to the catalog.

For a list of properties supported by **Role** collections,see the Using MTS Collections topic.

See Also

AssociateRoleByName Method (RoleAssociationUtil)

AssociateRoleByName Method (RoleAssociationUtil)

The **AssociateRoleByName** method associates a role with a specified component or component interface.

Syntax

object.**AssociateRoleByName(***name***)**

Parameters

object
: Required. An object variable that evaluates to a **RoleAssociationUtil** utility object.

name
: Required; **String**. An expression providing the name of the role to associate with a component or component interface.

Remarks

The changes are applied immediately to the catalog. For a list of properties supported by **Role** collections,see the Using MTS Collections topic.

See Also

AssociateRole Method (RoleAssociationUtil)

Connect Method (Catalog)

Connects to a catalog and returns a root collection.

Syntax

set root object.**Connect(***name***)**

Parameters

root
Required. String containing the root collection that serves as a starting point to locate top-level collections.

object
Required. An object variable that evaluates to a catalog object.

name
Required; **String**. String containing the name of a remote computer. To connect to a local computer, supply an empty string as this argument.

Remarks

The **Connect** method returns a root collection, which is bound to the connected computer. A root collection serves as a starting point to locate top-level collections, and does not contain any objects or properties.

Note that you can also use the **GetCollection** method to locate a package on the local computer without first having to call the **Connect** method.

You can get the following collections from the root collection:

Packages
RemoteComponents
RelatedCollectionInfo
ComputerList
LocalComputer
PropertyInfo

For a list of the MTS collections and their properties, see the Using MTS Collections topic.

See Also

GetCollection Method (CatalogCollection)

Count Property (CatalogCollection)

Returns an integer value indicating the number of objects in a collection.

Syntax

*object***.Count**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

Upon instantiation, a **CatalogCollection** object returns a count of zero. Call the **Populate** method to read from the **CatalogCollection** object, and then use the **Count** method to return the number of objects in the collection.

for a list of the MTS collections and their properties, see the Using MTS Collections topic.

See Also

Populate Method (CatalogCollection)

DataStoreMajorVersion Property (CatalogCollection)

Returns an integer value indicating the major version number of the catalog.

Syntax

object.**DataStoreMajorVersion**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

See Also

DataStoreMinorVersion Property (CatalogCollection),

DataStoreMinorVersion Property (CatalogCollection)

Returns an integer value indicating the minor version number of the catalog.

Syntax

object.**DataStoreMajorVersion**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

See Also

DataStoreMajorVersion Property (CatalogCollection)

ExportPackage Method (PackageUtil)

Exports a package.

Syntax

*object***.ExportPackage(***PackID, FileName, Options***)**

Parameters

object
  Required. An object variable that evaluates to a **PackageUtil** object.

PackID
  Required; **String**. An object variable that specifies the package ID of the package to export.

FileName
  Required; **String**. An object variable that provides the name of the package file to export.

Options
  Required; **Long**. An integer value specifying export options. This parameter can be 0 or MtsExportUsers, which includes users in roles in the package file.

GetCLSIDs Method(ComponentUtil)

Returns an array of installable class identifiers (CLSIDs) in the component DLL and/or type library.

Syntax

*object***.GetCLSIDs(***bstrDLLFile, bstrTypeLibFile, aCLSIDs***)**

Parameters

BstrDLLFile
: Required; **String**. A string variable that evaluates to the path of the DLL that you want checked.

BstrTypeLibFile
: Required; **String**. A string variable that evaluates to the path of the type library that you want checked. If the type library is embedded with the DLL (as is the case with DLLs generated by Microsoft Visual Basic), this parameter should be an empty string).

aCLSIDS
: Required; **Variant**. An output array of CLSIDs (VARIANTS) that can be installed from the supplied DLL and/or type library.

GetCollection Method (Catalog)

Instantiates a **CatalogCollection** object.

Syntax

set x object.**GetCollection(***Name***)**

Parameters

x
Required. An object variable (a variant, or object variable, or a **CatalogCollection** variable) for the returned collection.

object
Required. An object variable that evaluates to a catalog object.

Name
Required; **String**. A string expression containing the name of the collection to instantiate.

Remarks

You can use this method to get the following collections:

Packages
ComputerList
LocalComputer
RemoteComponents
RelatedCollectionInfo

After using the **GetCollection** method, you must fill the object by calling the **Populate** method. See the **Populate** method topic for further detail.

For a list of the MTS collections and their properties, see the Using MTS Collections topic..

GetCollection Method (CatalogCollection)

Retrieves a collection from the catalog.

Syntax

set x   object.**GetCollection(***name, key***)**

Parameters

x
: Required. An object variable (a variant, or object variable, or a **CatalogCollection** variable) for the returned collection.

object
: Required. An object variable that evaluates to a **CatalogCollection** object.

name
: Required. A **String** expression containing the name of the collection to instantiate.

key
: Required. A **Variant** expression containing the key of the object from which to navigate.

Remarks

Note that the **GetCollection** method gets an empty collection; you must call the **Populate** method to fill the collection.

For a list of the MTS collections and their properties, see the Using MTS Collections topic.

See Also

Populate Method (CatalogCollection),

GetUtilInterface Method (CatalogCollection)

Instantiates a utility object for the collection.

Syntax

set util *object***.GetUtilInterface**

Parameters

util
  Required. An object variable that evaluates to a catalog utility object.

object
  Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

Call the **GetUtilInterface** method to instantiate any one of the **PackageUtil**, **ComponentUtil**, **RemoteComponentUtil**, and **RoleAssociationUtil** utility objects. This method is only supported on **Packages**, **ComponentsInPackage**, **RemoteComponents**, **RolesForPackageComponent**, and **RolesForPackageComponentInterface** collections.

For a list of the MTS collections and their properties, see the Using MTS Collections topic.

ImportComponent Method (ComponentUtil)

Imports a component that is already registered as an in-process (in-proc) server.

Syntax

object.**ImportComponent(***CLSID***)**

Parameters

object
  Required. An object variable that evaluates to a **ComponentUtil** object.

CLSID
  Required; **String**. An expression containing the CLSID of the component to be installed.

Remarks

The changes are applied immediately to the catalog.

For a description of the **Component** collection and its associated properties, see the Using MTS Collections topic.

See Also

ImportComponentByName Method (ComponentUtil)

ImportComponentByName Method (ComponentUtil)

Imports a component that is already registered as an in-process server by the component's programmatic identifier (ProgID).

Syntax

object.**ImportComponentByName(***ProgID***)**

Parameters

object
  Required. An object variable that evaluates to a **ComponentUtil** object.
ProgID
  Required. A **String** expression identifying the ProgID of the component to be installed.

Remarks

The changes are applied immediately to the catalog. For a description of the **Component** collection and its associated properties, see the Using MTS Collections topic.

See Also

ImportComponent Method (ComponentUtil)

InstallComponent Method (ComponentUtil)

Installs a component into a package.

Syntax

*object***.InstallComponent(***filepath, typelibrary, proxy-stub***)**

Parameters

object
  Required. An object variable that evaluates to a **ComponentUtil** object.

filepath
  Required. A **String** expression that provides the file path of the DLL containing the components to install.

typelibrary
  Required. A **String** expression that provides the file path of the type library to use. Pass an empty string as this argument if the type library is embedded in the DLL.

proxy-stub
  Required. A **String** expression that provides the file path of a custom proxy-stub DLL to use. Pass an empty string as this argument if there is no custom proxy-stub DLL.

Remarks

The changes are applied immediately to the catalog.

For a description of the **Components** collection and its associated properties, see the Using MTS Collections topic.

See Also

InstallRemoteComponent Method (RemoteComponentUtil), **InstallRemoteComponentByName Method (RemoteComponentUtil)**

InstallPackage Method (PackageUtil)

Installs a component or components that are valid within a package's collection.

Syntax

*object***.InstallPackage(***FileName, InstallPath, options***)**

Parameters

object
   Required. An object variable that evaluates to a **Package** utility object.

FileName
   Required; **String**. String expression evaluating to the name of the package to install.

InstallPath
   Required; **String**. String expression evaluating to the install path for component files.

options
   Required; **Long**. A long value specifying install options. This parameter can be 0 or mtsInstallUsers, which adds users saved in the package file. If this option is not specified, users saved in the package file are not installed.

Remarks

All component files must be in the same directory of the package file. Component files arecopied to the install path specified as an argument of the **InstallPackage** method.

The changes are applied immediately to the catalog.

For a description of the **Components** collection and its associated properties, see the Using MTS Collections topic..

InstallRemoteComponent Method (RemoteComponentUtil)

Pulls remote components from a package on a remote server.

Syntax

*object***.InstallRemoteComponent(***computer, PackID, CLSID***)**

Parameters

object
: Required. An object variable that evaluates to **a RemoteComponentUtil** object.

computer
: Required; **String**. A string expression providing the name of the remote computer.

PackID
: Required; **String**. A string expression providing the package identification of the package containing the remote component.

CLSID
: Required; **String**. A string expression containing the CLSID of the remote component.

Remarks

The changes are applied immediately to the catalog.

See the Working with Remote MTS Computers topic for a complete description of how to pull components from a remote server.

See Also

InstallRemoteComponentByName Method (RemoteComponentUtil)

InstallRemoteComponentByName Method (RemoteComponentUtil)

Pulls remote components by name from the package on a remote server.

Syntax

object.**InstallRemoteComponentByName(***computer, PackName, ProgID***)**

Parameters

object
Required. An object variable that evaluates to a **RemoteComponentUtil** object.

computer
Required; **String**. A string expression providing the name of the remote computer.

PackName
Required; **String**. A string expression providing the name of the package containing the remote component.

ProgID
Required; **String**. A string value containing the ProgID of the remote component.

Remarks

The changes are applied immediately to the catalog. See the Working with Remote MTS Computers topic for a complete description of how to pull components from a remote server.

See Also

InstallRemoteComponent Method (RemoteComponentUtil)

IsPropertyReadOnly Property (CatalogObject)

Returns a **Boolean** value that indicates if the property for an object is set to read-only.

Syntax

object.**IsPropertyReadOnly(***value***)**

Parameters

object
  Required. An object variable that evaluates to a catalog object property.
value
  Required. The name of the value for which to check the read-only property.

Remarks

If the value returned by the **IsPropertyReadOnly** method is **True**, then you cannot modify the property. If the value returned is **False**, you can modify the property using the **Value** property.

IsPropertyWriteOnly Property (CatalogObject)

Returns a **Boolean** value that indicates if the property for an object is set to write-only.

Syntax

object.**IsPropertyWriteOnly(***propertyname***)**

Parameters

object
Required. An object variable that evaluates to a catalog object property.

propertyname
Required. The name of the property for which to check the write-only status.

Remarks

If the value returned by the **IsPropertyWriteOnly** method is **True**, then you can write but not read the property value. If the value returned is **False**, you can read the property value.

See Also

IsPropertyReadOnly Property (CatalogObject)

Item Property (CatalogCollection)

Gets a specific object in a collection.

Syntax

*object***.Item(***index***)**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

index
Required; **Long**. A zero-based index that specifies the position of a member of the collection. Must be a number from 0 through the value of the collection's **Count** property -1.

Key Property (CatalogObject)

Gets the value of the key of the object.

Syntax

object.**Key**

Parameters

object
Required. An object variable that evaluates to a **CatalogObject** object.

Remarks

All catalog objects have a key. The object key is a single property that uniquely identifies the object. To access a related collection using the **GetCollection** method, provide the key of the object from which you want to navigate (such as the package identifier). The following table provides the key property for each collection supported:

Collection	Key Property
Packages	ID
ComponentsInPackage	CLSID
RolesInPackage	ID
RolesForPackageComponent	ID
RolesForPackageComponentInterface	ID
UsersInRole	User
InterfacesForComponent	IID
InterfacesForRemoteComponent	IID
RemoteComponents	CLSID
MethodsForInterface	Name

MajorVersion Property (Catalog)

Returns an integer value indicating the major version number of the catalog.

Syntax

object.**MajorVersion**

Parameters

object
Required. An object variable that evaluates to a catalog object.

See Also

MinorVersion Property (Catalog)

MinorVersion Property (Catalog)

Returns an integer value indicating the minor version number of the catalog.

Syntax

object.**MinorVersion**

Parameters

object
Required. An object variable that evaluates to a catalog object.

See Also

MajorVersion Property (Catalog)

Name Property (CatalogObject)

Gets the name of an object.

Syntax

object.**Name**

Parameters

object
Required. An object variable that evaluates to a catalog object.

Remarks

All catalog objects have a name property. The following table provides the name property for each collection supported:

Collection	Name Property
Packages	Name
ComponentsInPackage	ProgID
RolesInPackage	Name
RolesForPackageComponent	Name
RolesForPackageComponentInterface	Name
UsersInRole	User
InterfacesForComponent	Name
InterfacesForRemoteComponent	Name
RemoteComponents	ProgID
MethodsForInterface	Name

Name Property (CatalogCollection)

Gets the name of the collection.

Syntax

object.**Name**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

See the Using MTS Collections topic for a list of properties supported by each MTS collection.

Populate Method (CatalogCollection)

Fills a collection with objects from the catalog.

Syntax

object.**Populate**

Parameters

object
Required. An object variable that evaluates to the **CatalogCollection** object that you would like to fill.

Remarks

The **Populate** method reads the contents of the **CatalogCollection** object. Any changes that are still pending (such as property changes, objects added, or objects removed) are lost. See the **SaveChanges** method topic for instruction on how to preserve changes made to a **CatalogCollection** object.

PopulateByKey Method (CatalogCollection)

Populates the collection with information for the specified objects.

Syntax

object.**PopulateByKey(aCLSIDs)**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

aCLSIDS
Required; **Variant**. An array of object keys (VARIANTS) denoting which objects should have their information read from the catalog.

Remove Method (CatalogCollection)

Removes an item from an object given the index position.

Syntax

object.**Remove(***index***)**

Parameters

object
  Required; **String**. An object variable that evaluates to a **CatalogCollection** object.
index
  Required; **Long**. A zero-based index indicating the position of the object to remove.

Remarks

The object is removed from the collection and all objects with higher indices are shifted up. Note that the **Count** property of a collection changes after the **Remove** method has been called.

Call the **SaveChanges** method to save the changes made to the collection using the **Remove** method.

RemoveEnabled Property (CatalogCollection)

Returns a Boolean value indicating that you can use the **Remove** method to delete an object from the collection.

Syntax

object.**RemoveEnabled**

Parameters

object
Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

If the value returned is **True**, then you can call the **Remove** method to remove an object from the collection. If the value returned is **False**, objects cannot be removed from the collection.

SaveChanges Method (CatalogCollection)

Saves changes to a collection in the catalog, and returns an integer indicating the number of changes applied to the collection.

Syntax

object.**SaveChanges**

Parameters

object
  Required. An object variable that evaluates to a **CatalogCollection** object.

Remarks

The **SaveChanges** method works exclusively on the collection on which you call it, and applies all pending changes to the catalog. If no changes are pending, then the method returns zero.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ShutdownPackage Method (PackageUtil)

Initiates the shutting down of a package. Shutting down a package terminates that application's process.

Syntax

object.**ShutdownPackage(bstrPackageID)**

Parameters

object
  Required. An object variable that evaluates to a **PackageUtil** object.

BstrPackageID
  Required. A string variable that evaluates to the **PackageID** of a **Package CatalogObject**.

Remarks

The **ShutdownPackage** method shuts down a single package process.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

Valid Property (CatalogObject)

Returns a Boolean value indicating whether all the object properties were successfully read from the catalog.

Syntax

object.**Valid**

Parameters

object
Required. An object variable that evaluates to a **CatalogObject**.

Remarks

If this property is **False** it indicates that one or more object properties could not be read from the catalog during a call to **Populate**. This property will be **True** for objects that have been added to the collection using the **Add** method.

Value Property (CatalogObject)

Gets or sets a value for an object property.

Syntax

object.**Value(***property***)**   *value*

Parameters

object
  Required. An object variable that evaluates to a **CatalogObject**.

property
  Required. A **String** expression of any type that specifies the name of the property whose value to get or set.

value
  Required. A **String** expression that specifies the value of the property to get or set.

Remarks

See the Using MTS Collections topic for a description the properties supported by the MTS administration objects.

Automating MTS Administration With Visual C++

The topics in this section describe administration interfaces supported by Microsoft Transaction Server (MTS). This reference describes the following topics:

MTS Visual C++ Error Codes
ICatalog
ICatalogObject
ICatalogCollection
IPackageUtil
IComponentUtil
IRemoteComponentUtil
IRoleAssociationUtil

The **ICatalog**, **ICatalogObject**, and **ICatalogCollection** interfaces provide top-level functionality such as creating and modifying objects. The **ICatalog** interface enables you to connect to specific servers and access collections. Call the **ICatalogCollection** interface to enumerate, create, delete, and modify objects, as well as to access related collections. The **ICatalogObject** interface is used to get and set properties on an object.

The utility interfaces (**IRemoteComponentUtil** and **IRoleAssociationUtil**) allows you to program very specific tasks for collection types, such as associating a role with a user or class of users.

MTS Visual C++ Error Codes

The following is a list of the error codes returned by methods called on the catalog collection and catalog utility interfaces.

E_INVALIDARG
Indicates one of the following:

- An invalid collection or property name was entered.
- An out parameter was NULL.
- The value is not one of the supported values or falls outside the supported range.
- The property is read-only.
- The property cannot be changed after the object is created.
- An invalid index was entered.

E_NOTIMPL
Object has been removed from the collection and is not supported on this collection.

E_MTS_OBJECTERRORS
Errors were encountered processing some objects or file. See the **ErrorInfo** collection for object/file-specific error codes.

E_MTS_NOUSER
User ID for user in role is not valid.

E_MTS_USERPASSWDNOTVALID
Package identity user ID and/or password are not valid

E_MTS_AUTHENICATIONLEVEL
Required authentication level (package privacy) could not be set for package updates.

E_MTS_PDFREADFAIL
An error occurred reading the package file.

E_MTS_PDFVERSION
Package file version is invalid.

E_MTS_BADPATH
Package file path is invalid.

E_MTS_PACKAGEEXISTS
Package with the same ID is already installed.

E_MTS_ROLEEXISTS
A role with the same ID is already installed. The role ID in the package file is likely corrupted.

E_MTS_CANTCOPYFILE
Errors occurred copying one or more files to the install directory.

E_MTS_INVALIDUSERIDS
One or more user IDS for roles were invalid.

E_MTS_CLSIDORIIDMISMATCH
One or more component/interface identifiers in a component DLL does not match the identifiers saved in the package file. The package file is out of date.

E_MTS_PACKDIRNOTFOUND
Package install directory is invalid due to general registry read/write errors.

E_MTS_PDFWRITEFAIL
An error occurred writing the package file.

E_MTS_NOTYPELIB
Could not find the type library for one or more components.

The following is a list of the object or file-specific error codes returned in **ErrorInfo** collections:

E_OBJECTINVALID

One or more object properties is corrupted or invalid.

E_KEYMISSING
One or more objects is not found in the catalog.

E_ALREADYINSTALLED
Component is already installed.

E_DOWNLOADFAILLED
One or more component files could not be copied to the client.

E_REMOTEINTERFACE
No interface information is available for the component. Component files could not be downloaded.

E_COREQCOMPINSTALLED
Component in the same DLL file is already installed.

E_NOREGISTRYCLSID
Component's CLSID is corrupted.

E_BADREGISTRYPROGID
Component's ProgID is corrupted.

E_DLLLOADFAILED
Component's DLL could not be loaded.

E_DLLREGISTERSERVER
DllRegisterServer method failed during component self-registration.

E_NOSERVERSHARE
No file share is available on the server to copy component files from the network path.

E_NOACCESSTOUNC
Network path registered for this component could not be accessed.

E_BADREGISTRYLIBID
Component type library ID is corrupted.

E_TREATAS
Component **TreatAs** key was found, but is not supported.

E_BADFORWARD
IID forward entry is corrupted.

E_BADIID
IID is corrupted.

E_REGISTRARFAILED
Component registrarmethod failed during component install.

E_COMFILE_DOESNOTEXIST
Component file does not exist.

E_COMFILE_LOADDLLFAIL
DLL file could not be loaded.

E_COMFILE_GETCLASSOBJ
DllGetClassObject method call failed during the DLL self-registration process.

E_COMFILE_CLASSNOTAVAIL
Class coded in the type library was not supported.

E_COMFILE_BADTLB
Type library was corrupted.

E_COMFILE_NOTINSTALLABLE
File does not contain COM components or type library information.

The following is a list of general read and write registry errors:

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE

Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

E_MTS_NOTCHANGEABLE
Changes to this object and sub-objects have been disabled.

E_MTS_NOTDELETABLE
Delete function for this object has been disabled.

E_MTS_SESSION
Server catalog version is not supported.

MTS ICatalog Interface

The **Catalog** object enables you to connect to specific servers and access collections. The **ICatalog** interface contains the following methods:

- **ICatalog::GetCollection**
- **ICatalog::Connect**
- **ICatalog::get_MajorVersion**
- **ICatalog::get_MinorVersion**

ICatalog::GetCollection

The **GetCollection** method retrieves a local collection without reading any objects from the catalog.

Syntax

HRESULT ICatalog::GetCollection(
BSTR *bstrCollName*
IDispatch ** *ppCatalogCollection***);**

Parameters

bstrCollName [in]
BSTR containing the name of the collection to retrieve from the catalog.

ppCatalogCollection [out]
Pointer to a pointer to the **CatalogCollection** object.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Invalid collection name passed as a parameter. Can also indicate that an *Out* parameter is NULL.

ICatalog::Connect

The **Connect** method connects to a remote computer and returns a *root collection*, which is bound to a remote computer.

HRESULT ICatalog::Connect(
BSTR *bstrCollName*
IDispatch ** *ppCatalogCollection***);**

Parameters

bstrConnectString [in]
BSTR expression containing the name of a remote computer.

PpCatalogCollection [out]
Pointer to a pointer to the **CatalogCollection** object.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

Comments

A root collection serves as a starting point to locate packages, and contains neither objects nor properties. Note that you can also use the **GetCollection** method to get a top-level collection on a local server without using the **Connect** method.

ICatalog::get_MajorVersion

The **get_MajorVersion** method returns the major version number of an administration object.

HRESULT ICatalog::get_MajorVersion(
long* *retval***);**

Parameters

retval [out]
Pointer to the major version number of the MTS object.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

Comments

Call the **get_MajorVersion** method and the **get_MinorVersion** method to determine if you are using the most current version of MTS

ICatalog::get_MinorVersion

The **get_MinorVersion** method retrieves the minor version number of an MTS administration object.

HRESULT ICatalog::get_MinorVersion(
  long* *retval*
  );

Parameters

retval [out]
  Pointer to the minor version number of the MTS object.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out parameter is NULL.

Comments

Call the **get_MajorVersion** method and the **get_MinorVersion** method to determine if you are using the most current version of MTS

MTS ICatalogCollection Interface

The **CatalogCollection** object can be used to enumerate objects, create, delete, and modify objects, and access related collections. The **ICatalogCollection** interface contains the following methods:

- **ICatalogCollection::get_NewEnum**
- **ICatalogCollection::get_Item**
- **ICatalogCollection::get_Count**
- **ICatalogCollection::Remove**
- **ICatalogCollection::Add**
- **ICatalogCollection::Populate**
- **ICatalogCollection::SaveChanges**
- **ICatalogCollection::GetCollection**
- **ICatalogCollection::get_Name**
- **ICatalogCollection::get_AddEnabled**
- **ICatalogCollection::get_RemoveEnabled**
- **ICatalogCollection::GetUtilInterface**
- **ICatalogCollection::get_DataStoreMajorVersion**
- **ICatalogCollection:: get_DataStoreMinorVersion**
- **ICatalogCollection::PopulateByKey**

ICatalogCollection::get_NewEnum

The **get_NewEnum** method returns the **IEnumVariant** enumerator interface.

HRESULT ICatalogCollection::get_NewEnum(
IUnknown** *ppEnumVariant***);**

Parameters

ppEnumVariant [out]
Pointer to a pointer to the **IEnumVariant** interface.

Return Values

S_OK
Method completed successfully.

INVALIDARG
Out parameter is NULL.

ICatalogCollection::get_Item

The **get_Item** method returns an object from the collection represented by the index.

HRESULT ICatalogCollection::get_Item(
  long *1Index*
  IDispatch** *ppCatalogObject***);**

Parameters

1Index [in]
  Index to the object in the collection.

PpCatalogObject [out]
  Pointer to a pointer to the **Catalog** object.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out of range of index.

Comments

A collection object contains zero or more items (all MTS collections are zero-based).

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::get_Count

The **get_Count** method returns the number of objects in the collection.

HRESULT ICatalogCollection::get_Count(
long* *retval***);**

Parameters

retval [out]
Pointer to the number of elements in the object collection.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::Add

The **Add** method adds a default object to the collection and returns a pointer to the new object.

HRESULT ICatalogCollection::Add(
IDispatch** *ppCatalogObject***);**

Parameters

ppCatalogObject [out]
Pointer to a pointer to the new **Catalog** object.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

E_NOTIMPL
Not supported on this collection.

Remarks

To update the objects in a collection, call the **Add** method to create a new object either before or after populating a collection.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::Populate

The **Populate** method reads the collection objects from the catalog.

HRESULT ICatalogCollection::Populate( );

Return Values

S_OK
Method completed successfully.

REGDB_E_CLASSNOTREG
The **MTXCatEx.CatalogServer.1** component is not registered on the target computer. MTS is not installed properly on the target computer.

E_MTS_OBJECTERRORS
Collection was read but some objects were invalid. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

Remarks

You call the **Populate** method to fill a package collection with objects from the catalog. This method uses the **CoCreateInstance** function internally, so **CoCreateInstance** error codes are included in the **Populate** method's return values.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::SaveChanges

The **SaveChanges** method saves changes to the collection into the catalog.

HRESULT ICatalogCollection::SaveChanges(
long* *retval***);**

Parameters

retval [out]
Number of changes applied to the collection.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

E_MTS_OBJECTERRORS
Errors were encountered processing some objects. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOUSER
User ID is invalid.

E_MTS_USERPASSWDNOTVALID
Package identity user ID and/or password are invalid.

E_MTS_AUTHENTICATIONLEVEL
Required authentication level (package privacy) could not be set for package updates.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

CLASS_E_NOAGGREGATION
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Note that you must call this method after modifying any object in the collection. The **SaveChanges** method works exclusively on the collection on which it is called. This method also applies all pending changes to objects within a given collection. This method uses the **CoCreateInstance** function internally, so **CoCreateInstance** error codes are included in the **SaveChanges** method's return values.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::GetCollection

The **GetCollection** method retrieves a collection related to a specific object. Data is not read from the catalog. See the **Populate** method topic for more information.

HRESULT ICatalogCollection::GetCollection(
  BSTR *bstrCollName*
  VARIANT *varObjectKey*
  IDispatch** *ppCatalogCollection***);**

Parameters

bstrCollName [in]
  BSTR containing the name of the collection.

VarObjectKey [in]
  Value of the object key.

retval [out]
  Pointer to a pointer to the **CatalogCollection** object.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Invalid collection name passed as a parameter. Can also indicate that an *Out* parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::get_Name

The **get_Name** method gets the name of a collection.

HRESULT ICatalogCollection::get_Name(
  VARIANT* *retval***);**

Parameters

retval [out]
  Pointer to the name of the collection.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::get_AddEnabled

The **get_AddEnabled** method returns a value that indicates if the **Add** method is supported in this collection.

HRESULT ICatalogCollection::get_AddEnabled(
VARIANT_BOOL* *varAddEnabled***);**

Parameters

VarObjectKey [out]
Boolean value indicating if the **Add** method is supported in this collection.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::Remove

The **Remove** method removes an item from a collection, given the index of the item.

HRESULT ICatalogCollection::Remove(
long *1Index***);**

Parameters

1Index [in]
Index position of the object to remove.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Invalid index was entered.

E_NOTIMPL
Collection does not support removing objects.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::get_RemoveEnabled

The **get_RemoveEnabled** method returns a value that indicates if the **Remove** method is supported in this collection.

HRESULT ICatalogCollection:: get_RemoveEnabled(
VARIANT_BOOL* *boolRemoveEnabled***);**

Parameters

boolRemoveEnabled [out]
Boolean value indicating if the **Remove** method is supported in this collection.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::GetUtilInterface

The **GetUtilInterface** method returns a pointer to the interface of the utility object for a package, component, remote component, or role collection.

HRESULT ICatalogCollection::GetUtilInterface(
  IDispatch** *ppUtil***);**

Parameters

ppUtil [out]
  Pointer to a pointer to the interface on a utility object.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out parameter is NULL.

E_NOTIMPL
  Not supported on this collection.

Remarks

Call the **GetUtilInterface** method to program your application for specific administration tasks, such as creating a package or installing a component.

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogCollection::get_DataStoreMajorVersion

The **get_DataStoreMajorVersion** method returns the major version number of the catalog from which you get the collection.

HRESULT ICatalogCollection::get_DataStoreMajorVersion(
long* *retval***);**

Parameters

retval [out]
Pointer to a pointer to the MTS major version number.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

See Also

get_DataStoreMinorVersion

ICatalogCollection::get_DataStoreMinorVersion

The **get_DataStoreMinorVersion** method returns the minor version number of the catalog from which you get a collection.

HRESULT ICatalogCollection:: get_DataStoreMinorVersion(
long* *retval***);**

Parameters

retval [out]
Pointer to a pointer to the MTS minor version number.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

See Also

get_DataStoreMajorVersion

ICatalogCollection::PopulateByKey

The **PopulateByKey** method populates the collection with information for the specified objects.

HRESULT ICatalogCollection::PopulateByKey(
  SAFEARRAY* *saKeys***);**

Parameters

saKeys [in]
  Pointer to a safearray of **VARIANTS** containing the CLSIDs of components for which the collection object should refresh its information.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out parameter is NULL.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

MTS ICatalogObject Interface

The **CatalogObject** object provides methods to get and set properties on an object. The **ICatalogObject** interface contains the following methods:

- **ICatalogObject::get_Value**
- **ICatalogObject::put_Value**
- **ICatalogObject::get_Key**
- **ICatalogObject::get_Name**
- **ICatalogObject::IsPropertyReadOnly**
- **ICatalogObject::IsPropertyWriteOnly**
- **ICatalogObject::get_Valid**

ICatalogObject::get_Value

The **get_Value** method gets a property value of an object in a collection.

HRESULT ICatalogObject::get_Value(
  BSTR *bstrPropName*
  VARIANT* *retval***);**

Parameters

bstrPropName [in]
  BSTR expression containing the name of the property.

retval [out]
  Pointer to the value of the property.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
  Invalid property name passed as a parameter. Can also indicate that an *Out* parameter is NULL.

E_NOTIMPL
  Object has been removed from the collection.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogObject::put_Value

The **put_Value** method sets the property value of an object in a collection.

HRESULT ICatalogObject::put_Value(
  BSTR *bstrPropName*
  VARIANT *val***);**

Parameters

bstrPropName [in]
  BSTR containing the name of the property to set.

val [in]
  Variant containing the new value for the property.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Invalid property name entered. Can also indicate either the property value is not one of the supported values or falls outside the supported range, the property is read-only, or the property cannot be changed after the object is created.

E_NOTIMPL
  Object has been removed from the collection.

Remarks

See the Using MTS Collections topic for a list of the MTS collections and their properties.

ICatalogObject::get_Key

The **get_Key** method gets the value of the **Key** property.

HRESULT ICatalogObject::get_Key(
VARIANT* *retval***);**

Parameters

retval [out]
Pointer to the value of the **Key** property.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

E_NOTIMPL
Object has been removed from the collection.

Comments

All MTS objects have a key. The object key is a single property that uniquely identifies the object. To create a related collection in your Package collection object, provide the key of the object from which you want to navigate (such as the package identifier). The following table provides the key property for each collection supported by the MTS Explorer.

Collection	Key Property
Packages	ID
ComponentsInPackage	CLSID
RolesInPackage	ID
RolesForPackageComponents	ID
UsersInRole	User
InterfacesForComponent	IID
InterfacesForRemoteComponent	IID

ICatalogObject::get_Name

The **get_Name** method provides the name of an object in the catalog.

HRESULT ICatalogObject::get_Name(
  VARIANT* *retval* **);**

Parameters

retval [out]
  Pointer to the name of the object.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Out parameter is NULL.

E_NOTIMPL
  Object has been removed from the collection.

All MTS objects have a name property. The following table provides the name property for each collection supported by the MTS Explorer:

Collection	Name Property
Packages	Name
ComponentsInPackage	ProgID
RolesInPackage	Name
RolesForPackageComponents	Name
UsersInRole	User
InterfacesForComponent	Name
InterfacesForRemoteComponent	Name
RemoteComponents	ProgID
MethodsForInterface	Name

ICatalogObject::IsPropertyReadOnly

The **IsPropertyReadOnly** method determines if a property is read-only.

HRESULT ICatalogObject::IsPropertyReadOnly(
  BSTR *bstrPropName*
  VARIANT_BOOL* *retval***);**

Parameters

bstrPropName [in]
  BSTR containing the name of the property.

retval [out]
  Boolean indicating if the property is read-only.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Invalid property name passed as a parameter. Can also indicate that an *Out* parameter is NULL.

Remarks

For more information about read-only property values and collections, see the Using MTS Collections topic.

See Also

IsPropertyWriteOnly

ICatalogObject::IsPropertyWriteOnly

The **IsPropertyWriteOnly** method indicates if a property can be written but not read.

HRESULT ICatalogObject::IsPropertyWriteOnly(
  BSTR *bstrPropName*
  VARIANT_BOOL* *retval***);**

Parameters

bstrPropName [in]
  BSTR containing the name of the property that may or may not be write-only.

retval [out]
  Boolean indicating if the property is write-only.

Return Values

S_OK
  Method completed successfully.

E_INVALIDARG
  Invalid property name passed as a parameter. Can also indicate that an *Out* parameter is NULL.

Remarks

For more information about read-only property values and collections, see the Using MTS Collections topic.

See Also

IsPropertyReadOnly

ICatalogObject::get_Valid

The **get_Valid** method determines if properties on an object were successfully read from the catalog.

HRESULT ICatalogObject::get_Valid(
VARIANT_BOOL* *retval***);**

Parameters

retval [out]
Boolean indicating if properties were successfully read. If this method returns **True**, all properties on an object were read from the catalog.

Return Values

S_OK
Method completed successfully.

E_INVALIDARG
Out parameter is NULL.

E_NOTIMPL
Object removed from the collection.

MTS IPackageUtil Interface

The **IPackageUtil** object enables a package to be installed and exported within the **Packages** collection. The **IPackageUtil** interface contains the following methods:

IPackageUtil::InstallPackage
IPackageUtil::ExportPackage
IPackageUtil::ShutdownPackage

See the Using MTS Collections topic for a list of the MTS collections and their properties.

IPackageUtil::InstallPackage

The **InstallPackages** method installs a pre-built package.

HRESULT IPackageUtil::InstallPackage(
  BSTR *bstrPackageFile*
  BSTR *bstrInstallPath*
  long *1Options***);**

Parameters

bstrPackageFile [in]
  BSTR containing the name of the package file to install.

bstrInstallPath [in]
  BSTR containing component install path.

1Options [in]
  Integer specifying export options. This method supports MtsExportUsers, which includes users in roles in the package file.

Return Values

S_OK
  Method completed successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing objects and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_PDFREADFAIL
  Error occurred reading the package file.

E_MTS_PDFVERSION
  Package file version is invalid.

E_MTS_BADPATH
  Package file path is invalid.

E_MTS_PACKAGEEXISTS
  Package with the same ID is already installed.

E_MTS_ROLEEXISTS
  Role with the same ID is already installed. The role ID in the package file is likely corrupted.

E_MTS_CANTCOPYFILE
  Errors occurred copying one or more files to the install directory.

E_MTS_INVALIDUSERIDS
  One or more user IDs for roles were invalid.

E_MTS_CLSIDORIIDMISMATCH
  One or more component/interface identifiers in a component DLL do not match the identifiers saved in the package file. The package file is likely out of date.

E_MTS_PACKDIRNOTFOUND
  The package install directory is invalid.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not

installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **InstallPackage** method's return values.

IPackageUtil::ExportPackage

The **ExportPackage** method exports a package according to its package identifier.

HRESULT IPackageUtil::ExportPackage(
  BSTR *bstrPackageID*
  BSTR *bstrPackageFile*
  long *1Options***);**

Parameters

bstrPackageID [in]
  BSTR containing the unique identifier of the package to export.

bstrPackageFile [in]
  BSTR containing the name of the package file to export.

1Options [in]
  Either zero (for no option selected) or MtsExportUsers, which includes users in roles in the package file.

Return Values

S_OK
  Method completed successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing objects and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_PDFWRITEFAIL
  Error occurred writing the package file.

E_MTS_NOTYPELIB
  Could not find the type library for one or more components.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **ExportPackage** method's return values.

IPackageUtil::ShutdownPackage

The **ShutdownPackage** method shuts down a single package, thereby terminating that application process.

HRESULT IPackageUtil::ShutdownPackage(
BSTR *bstrPackageID*

Parameters

bstrPackageID [in]
BSTR containing the unique identifier of the package to shut down.

Return Values

S_OK
Method completed successfully.

E_MTS_OBJECTERRORS
Errors were encountered processing objects and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_PDFWRITEFAIL
Error occurred writing the package file.

E_MTS_NOTYPELIB
Could not find the type library for one or more components.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

MTS IComponentUtil Interface

The **IComponentUtil** object provides methods to install a component in a specific collection and to import components registered as an in-proc server. The **IComponentUtil** interface contains the following methods:

IComponentUtil::InstallComponent
IComponentUtil::ImportComponent
IComponentUtil::ImportComponentByName
IComponentUtil::GetCLSIDs

IComponentUtil::InstallComponent

The **InstallComponent** method installs a component.

HRESULT IComponentUtil::InstallComponent(
  BSTR *bstrDLLFile*
  BSTR *bstrTypelibFile*
  BSTR *bstrProxyStubDLL***);**

Parameters

bstrDLLFile [in]
  BSTR containing the name of the DLL file providing the components to install.

bstrTypelibFile [in]
  BSTR containing the name of the external type library file. If the type library file is embedded in the DLL, pass in an empty string for this parameter.

bstrProxyStubDLL [in]
  BSTR containing the name of the proxy-stub DLL file. If there is no proxy-stub DLL associated with the component, pass in an empty string for this parameter.

Return Values

S_OK
  Method completed successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **InstallComponent** method's return values.

IComponentUtil::ImportComponent

The **ImportComponent** method imports a component that is already registered as an in-process (in-proc) server.

```
HRESULT IComponentUtil::ImportComponent(
   BSTR                    bstrCLSID);
```

Parameters

bstrCLSID [in]
BSTR containing the CLSID of the component to import.

Return Values

S_OK
Method completed successfully.

E_MTS_OBJECTERRORS
Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **ImportComponent** method's return values.

See Also

ImportComponentByName

IComponentUtil::ImportComponentByName

The **ImportComponentByName** method imports a component that is already registered as an in-process (in-proc) server. This method uses the component's programmatic identifier (ProgID) for the import procedure.

HRESULT IComponentUtil::ImportComponentByName(
BSTR *bstrProgID***);**

Parameters

bstrProgID [in]
BSTR containing the **ProgID** of the component to import.

Return Values

S_OK
Method completed successfully.

E_MTS_OBJECTERRORS
Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **ImportComponentByName** method's return values.

See Also

ImportComponent

IComponentUtil::GetCLSIDs

The **GetCLSIDs** method fills an array with the installable component CLSIDs from a DLL and/or type library.

HRESULT IComponentUtil::GetCLSIDs(
  BSTR *bstrDLLFile*
  BSTR *bstrTypeLibFile*
  SAFEARRAY** *ppsaCLSIDs)*

Parameters

bstrDLLFile [in]
  BSTR containing the name of the DLL file providing the components to check for allowable installation.

bstrTypeLibFile [in]
  BSTR containing the name of the external type library file to check for installable components.

ppsaCLSIDs [out]
  Pointer to a pointer to a SAFEARRAY containing VARIANTS which contain the CLSIDs of installable components in the given DLL and/or type library.

Return Values

S_OK
  Method completed successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

MTS IRemoteComponentUtil Interface

You can use the **IRemoteComponentUtil** object to program your application to pull remote components from a package on a remote server. The **IRemoteComponentUtil** interface contains the following methods:

IRemoteComponentUtil::InstallRemoteComponent
IRemoteComponentUtil::InstallRemoteComponentByName

IRemoteComponentUtil::InstallRemoteComponent

The **InstallRemoteComponent** method pulls a component to install from a package on a remote server.

HRESULT IRemoteComponentUtil::InstallRemoteComponent(
BSTR *bstrServer*
BSTR *bstrPackageID*
BSTR *bstrCLSID***);**

Parameters

bstrServer [in]
BSTR containing the name of the remote server from which to pull the component to install.

PackageID [in]
BSTR containing the identifier of the package containing the remote component.

bstrCLSID [in]
BSTR containing the class identifier (CLSID) of the remote component.

Return Values

S_OK
Method returned successfully.

E_MTS_OBJECTERRORS
Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **InstallRemoteComponent** method's return values.

See Also

InstallRemoteComponentByName

IRemoteComponentUtil::InstallRemoteComponentByName

The **InstallRemoteComponentByName** method pulls remote components from the package on a remote server and installs the component by package name and programmatic ID (ProgID).

HRESULT IRemoteComponentUtil::InstallRemoteComponentByName(
  BSTR *bstrSever*
  BSTR *PackageName*
  BSTR *bstrProgID***);**

Parameters

bstrSever [in]
  BSTR containing the name of the remote server from which to pull the component to install.

PackageName [in]
  BSTR containing the name of the package containing the remote component.

bstrProgID [in]
  BSTR containing the ProgID of the component to install.

Return Values

S_OK
  Method completed successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing objects. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **InstallRemoteComponentByName** method's return values.

See Also

InstallRemoteComponent

MTS IRoleAssociationUtil Interface

Call methods on the **IRoleAssociationUtil** object to associate roles with a component or component interface. The **IRoleAssociationUtil** interface contains the following methods:

IRoleAssociationUtil::AssociateRole
IRoleAssociationUtil::AssociateRoleByName

IRoleAssociationUtil::AssociateRole

The **AssociateRole** method associates a role with a component or component interface.

HRESULT IRoleAssociationUtil::AssociateRole(
  BSTR *bstrRoleID*
  );

Parameters

bstrRoleID [in]
  BSTR containing the ID of the role to associate with a component or component interface.

Return Values

S_OK
  Method returned successfully.

E_MTS_OBJECTERRORS
  Errors were encountered processing objects. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
  Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
  Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
  Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
  The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **AssociateRole** method's return values.

See Also

AssociateRoleByName

IRoleAssociationUtil::AssociateRoleByName

The **AssociateRoleByName** method associates a role by its name with a specified component or component interface.

HRESULT IRoleAssociationUtil::AssociateRoleByName(
BSTR *bstrRoleName***);**

Parameters

bstrRoleName [in]
BSTR containing the name of the role to associate with a component or component interface.

Return Values

S_OK
Method returned successfully.

E_MTS_OBJECTERRORS
Errors were encountered processing components and/or files. See the **ErrorInfo** collection for object-specific error codes.

E_MTS_NOREGISTRYREAD
Access control failure reading a registry key.

E_MTS_NOREGISTRYWRITE
Access control failure writing a registry key.

E_MTS_NOREGISTRYREPAIR
Access control failure writing a registry key.

REGDB_E_CLASSNOTREG
The MTXCatEx.CatalogServer.1 component is not registered on the target computer. MTS is not installed properly on the target computer.

Remarks

Because this method uses the **CoCreateInstance** function internally, **CoCreateInstance** error codes are included in the **AssociateRoleByName** method's return values.

See Also

AssociateRole

GetObjectContext

Visual Basic	**GetObjectContext** Function
Visual C++	**GetObjectContext** Function
Visual J++	**MTx.GetObjectContext** Method

SafeRef

Visual Basic	**SafeRef** Function
Visual C++	**SafeRef** Function
Visual J++	**MTx.SafeRef** Method

IObjectContext Interface, ObjectContext Object

Visual Basic	**ObjectContext** Object
Visual C++	**IObjectContext** Interface
Visual J++	**IObjectContext** Interface

SetAbort Method

Visual Basic	**SetAbort** Method
Visual C++	**IObjectContext::SetAbort** Method
Visual Basic	**IObjectContext.SetAbort** Method

SetComplete Method

Visual Basic	**SetComplete** Method
Visual C++	**IObjectContext::SetComplete** Method
Visual J++	**IObjectContext.SetComplete** Method

EnableCommit Method

Visual Basic	**EnableCommit** Method
Visual C++	**IObjectContext::EnableCommit** Method
Visual J++	**IObjectContext.EnableCommit** Method

DisableCommit Method

Visual Basic	**DisableCommit** Method
Visual C++	**IObjectContext::DisableCommit** Method
Visual J++	**IObjectContext.DisableCommit** Method

CreateInstance Method

Visual Basic **CreateInstance** Method

Visual C++ **IObjectContext::CreateInstance** Method

Visual J++ **IObjectContext.CreateInstance** Method

IsInTransaction Method

Visual Basic	**IsInTransaction** Method
Visual C++	**IObjectContext::IsInTransaction** Method
Visual J++	**IObjectContext.IsInTransaction** Method

IsCallerInRole Method

Visual Basic	**IsCallerInRole** Method
Visual C++	**IObjectContext::IsCallerInRole** Method
Visual J++	**IObjectContext.IsCallerInRole** Method

IsSecurityEnabled Method

Visual Basic	**IsSecurityEnabled** Method
Visual C++	**IObjectContext::IsSecurityEnabled** Method
Visual J++	**IObjectContext.IsSecurityEnabled** Method

ITransactionContextEx Interface, TransactionContext Object

Visual Basic	**TransactionContext** Object
Visual C++	**ITransactionContextEx** Interface
Visual J++	**ITransactionContextEx** Interface

Abort Method

Visual Basic	**Abort** Method
Visual C++	**ITransactionContextEx::Abort** Method
Visual J++	**ITransactionContextEx.Abort** Method

Commit Method

Visual Basic	**Commit** Method
Visual C++	**ITransactionContextEx::Commit** Method
Visual J++	**ITransactionContextEx.Commit** Method

IObjectControl Interface

Visual Basic	**ObjectControl** Interface
Visual C++	**IObjectControl** Interface
Visual J++	**IObjectControl** Interface

Activate Method

Visual Basic	**Activate** Method
Visual C++	**IObjectControl::Activate** Method
Visual J++	**IObjectControl.Activate** Method

CanBePooled Method

Visual Basic	**CanBePooled** Method
Visual C++	**IObjectControl::CanBePooled** Method
Visual J++	**IObjectControl.CanBePooled** Method

Deactivate Method

Visual Basic	**Deactivate** Method
Visual C++	**IObjectControl::Deactivate** Method
Visual J++	**IObjectControl.Deactivate** Method

ISharedProperty Interface, SharedProperty Object

Visual Basic	**SharedProperty** Object
Visual C++	**ISharedProperty** Interface
Visual J++	**ISharedProperty** Interface

Value

Visual Basic	**Value** Property
Visual C++	**ISharedProperty::get_Value** Method
Visual C++	**ISharedProperty::put_Value** Method
Visual J++	**ISharedProperty.getValue** Method
Visual J++	**ISharedProperty.putValue** Method

ISharedPropertyGroup Interface, SharedPropertyGroup Object

Visual Basic	**SharedPropertyGroup** Object
Visual C++	**ISharedPropertyGroup** Interface
Visual J++	**ISharedPropertyGroup** Interface

CreateProperty Method

Visual Basic	**CreateProperty** Method
Visual C++	**ISharedPropertyGroup::CreateProperty** Method
Visual J++	**ISharedPropertyGroup.CreateProperty** Method

CreatePropertyByPosition Method

Visual Basic	**CreatePropertyByPosition** Method
Visual C++	**ISharedPropertyGroup::CreatePropertyByPosition** Method
Visual J++	**ISharedPropertyGroup.CreatePropertyByPosition** Method

Property

Visual Basic	**Property** Property
Visual C++	**ISharedPropertyGroup::get_Property** Method
Visual J++	**ISharedPropertyGroup.getProperty** Method

PropertyByPosition

Visual Basic	**PropertyByPosition** Property
Visual C++	**ISharedPropertyGroup::get_PropertyByPosition** Method
Visual J++	**ISharedPropertyGroup.getPropertyByPosition** Method

ISharedPropertyGroupManager Interface, SharedPropertyGroupManager Object

Visual Basic	**SharedPropertyGroupManager** Object
Visual C++	**ISharedPropertyGroupManager** Interface
Visual J++	**ISharedPropertyGroupManager** Interface

CreatePropertyGroup Method

Visual Basic	**CreatePropertyGroup** Method
Visual C++	**ISharedPropertyGroupManager::CreatePropertyGroup** Method
Visual J++	**ISharedPropertyGroupManager.CreatePropertyGroup** Method

Group

Visual Basic	**Group** Property
Visual C++	**ISharedPropertyGroupManager::get_Group** Method
Visual J++	**ISharedPropertyGroupManager.getGroup** Method

get_NewEnum, get__NewEnum Methods

Visual C++ **ISharedPropertyGroupManager::get__NewEnum** Method

Visual J++ **ISharedPropertyGroupManager.get_NewEnum** Method

IGetContextProperties Interface

Visual C++	**IGetContextProperties** Interface
Visual J++	**IGetContextProperties** Interface

Count Method

Visual Basic	**Count** Method
Visual C++	**Count** Method
Visual J++	**Count** Method

EnumNames Method

Visual C++ **EnumNames** Method

Visual J++ **EnumNames** Method

GetProperty Method

Visual C++ **GetProperty** Method

Visual J++ **GetProperty** Method

ISecurityProperty Interface

Visual Basic	**SecurityProperty** Object
Visual C++	**ISecurityProperty** Interface

GetDirectCallerName Method

Visual Basic **GetDirectCallerName** Method

Visual C++ **GetDirectCallerSID** Method

GetDirectCreatorName Method

Visual Basic	**GetDirectCreatorName** Method
Visual C++	**GetDirectCreatorSID** Method

GetOriginalCallerName Method

Visual Basic	**GetOriginalCallerName** Method
Visual C++	**GetOriginalCallerSID** Method

GetOriginalCreatorName Method

Visual Basic	**GetOriginalCreatorName** Method
Visual C++	**GetOriginalCreatorSID** Method

Post Method, Step1 (Visual Basic)

```
Public Function Post(ByRef lngAccount As Long, _
    ByRef lngAmount As Long) As String

    On Error GoTo ErrorHandler
    Post = "Hello from Account!!!"
    Exit Function
' Return the error message indicating that
' an error occurred.
ErrorHandler:
    Err.Raise Err.Number, "Bank.Account.Post", _
        Err.Description
End Function
```

Post Method, Step2 (Visual Basic)

```
Public Function Post(ByVal lngAccountNo As Long, _
    ByVal lngAmount As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' obtain the ADO environment and connection
    Dim adoConn As New ADODB.Connection
    Dim varRows As Variant

    adoConn.Open strConnect

    On Error GoTo ErrorCreateTable

    ' update the balance
    Dim strSQL As String
    strSQL = "UPDATE Account SET Balance = Balance + "_
        + Str$(lngAmount) + " WHERE AccountNo = "
        + Str$(lngAccountNo)

TryAgain:
    adoConn.Execute strSQL, varRows

    ' if anything else happens
    On Error GoTo ErrorHandler

    ' get resulting balance which may have been
    ' further updated via triggers
    strSQL = "SELECT Balance FROM Account " _
        + "WHERE AccountNo = " + Str$(lngAccountNo)

    Dim adoRS As ADODB.Recordset
    Set adoRS = adoConn.Execute(strSQL)
    If adoRS.EOF Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
            + Str$(lngAccountNo) + " not on file."
    End If

    Dim lngBalance As Long
    lngBalance = adoRS.Fields("Balance").Value

    ' check if account is overdrawn
    If (lngBalance) < 0 Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
            + Str$(lngAccountNo) _
            + " would be overdrawn by " _
            + Str$(lngBalance) + ". Balance is still "
            + Str$(lngBalance - lngAmount) + "."
    Else
        If lngAmount < 0 Then
            strResult = strResult _
                & "Debit from account "
```

```
                & lngAccountNo & ", "
        Else
            strResult = strResult _
                & "Credit to account "
                & lngAccountNo & ", "
        End If
        strResult = strResult + "balance is $"
            & Str$(lngBalance) & ". (VB)"
    End If

    ' cleanup
    Set adoRS = Nothing
    Set adoConn = Nothing

    Post = strResult

Exit Function

ErrorCreateTable:
    On Error GoTo ErrorHandler

    ' create the account table
    Dim objCreateTable As CreateTable
    Set objCreateTable = _
    GetObjectContext.CreateInstance("Bank.CreateTable")
    objCreateTable.CreateAccount

    GoTo TryAgain

ErrorHandler:
    ' cleanup
    If Not adoRS Is Nothing Then
        Set adoRS = Nothing
    End If
    If Not adoConn Is Nothing Then
        Set adoConn = Nothing
    End If

    Post = ""         ' indicate that an error occurred
    Err.Raise Err.Number, "Bank.Accout.Post", _
        Err.Description

End Function
```

Post Method, Step3 (Visual Basic)

```
Public Function Post(ByVal lngAccountNo As Long, _
    ByVal lngAmount As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' obtain the ADO environment and connection
    Dim adoConn As New ADODB.Connection
    Dim varRows As Variant

    adoConn.Open strConnect

    On Error GoTo ErrorCreateTable

    ' update the balance
    Dim strSQL As String
    strSQL = "UPDATE Account SET Balance = Balance + "_
        + Str$(lngAmount) + " WHERE AccountNo = "
        + Str$(lngAccountNo)

TryAgain:
    adoConn.Execute strSQL, varRows

    ' if anything else happens
    On Error GoTo ErrorHandler

    ' get resulting balance which may have been
    ' further updated via triggers
    strSQL = "SELECT Balance FROM Account " _
        + "WHERE AccountNo = " + Str$(lngAccountNo)

    Dim adoRS As ADODB.Recordset
    Set adoRS = adoConn.Execute(strSQL)
    If adoRS.EOF Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
            + Str$(lngAccountNo) + " not on file."
    End If

    Dim lngBalance As Long
    lngBalance = adoRS.Fields("Balance").Value

    ' check if account is overdrawn
    If (lngBalance) < 0 Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
            + Str$(lngAccountNo) _
            + " would be overdrawn by " _
            + Str$(lngBalance) + ". Balance is still "
            + Str$(lngBalance - lngAmount) + "."
    Else
        If lngAmount < 0 Then
            strResult = strResult _
                & "Debit from account "
```

```
                & lngAccountNo & ", "
        Else
            strResult = strResult _
                & "Credit to account "
                & lngAccountNo & ", "
        End If
        strResult = strResult + "balance is $"
            & Str$(lngBalance) & ". (VB)"
    End If

    ' cleanup
    Set adoRS = Nothing
    Set adoConn = Nothing

    ' we are finished and happy
    GetObjectContext.SetComplete

    Post = strResult

Exit Function

ErrorCreateTable:
    On Error GoTo ErrorHandler

    ' create the account table
    Dim objCreateTable As CreateTable
    Set objCreateTable = _
    GetObjectContext.CreateInstance("Bank.CreateTable")
    objCreateTable.CreateAccount

    GoTo TryAgain

ErrorHandler:
    ' cleanup
    If Not adoRS Is Nothing Then
        Set adoRS = Nothing
    End If
    If Not adoConn Is Nothing Then
        Set adoConn = Nothing
    End If

    GetObjectContext.SetAbort          ' we are unhappy

    Post = ""          ' indicate that an error occurred
    Err.Raise Err.Number, "Bank.Accout.Post", _
        Err.Description

End Function
```

Perform Method, Step4 (Visual Basic)

```
Public Function Perform(ByVal lngPrimeAccount As Long,_
    ByVal lngSecondAccount As Long, ByVal lngAmount _
    As Long, ByVal lngTranType As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' create the account object using our context
    Dim objAccount As Bank.Account
    Set objAccount = _
        GetObjectContext.CreateInstance("Bank.Account")

    If objAccount Is Nothing Then
        Err.Raise ERROR_NUMBER, _
        Description:="Could not create account object"
    End If

    ' call the post function based on the
    ' transaction type
    Select Case lngTranType

        Case 1
            strResult = objAccount.Post(lngPrimeAccount, 0 - lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 2
            strResult = objAccount.Post(lngPrimeAccount, lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 3
            Dim strResult1 As String, strResult2 As String
            ' do the credit
            strResult1 = objAccount.Post(lngSecondAccount, lngAmount)
            If strResult1 = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult1
            Else
                ' then do the debit
                strResult2 = objAccount.Post(lngPrimeAccount, 0 -
lngAmount)
                If strResult2 = "" Then
                ' debit failed
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult2
                Else
                    strResult = strResult1 + "  " + strResult2
                End If
            End If
```

```
        Case Else
            Err.Raise ERROR_NUMBER, _
                Description:="Invalid Transaction Type"

    End Select

    ' we are finished and happy
    GetObjectContext.SetComplete

    Perform = strResult

    Exit Function

ErrorHandler:

    GetObjectContext.SetAbort          ' we are unhappy

    Perform = ""       ' indicate that an error occured

    Err.Raise Err.Number, "Bank.MoveMoney.Perform", _
        Err.Description

End Function
```

GetNextReceipt Method, Step5 (Visual Basic)

```
Public Function GetNextReceipt() As Long

    On Error GoTo ErrorHandler

    ' If Shared property does not already exist
    ' it will be initialized
    Dim spmMgr As SharedPropertyGroupManager
    Set spmMgr = CreateObject("MTxSpm.SharedPropertyGroupManager.1")

    Dim spmGroup As SharedPropertyGroup
    Dim bResult As Boolean
    Set spmGroup = _
        spmMgr.CreatePropertyGroup("Receipt", _
        LockMethod, Process, bResult)

    Dim spmPropNextReceipt As SharedProperty
    Set spmPropNextReceipt = _
        spmGroup.CreateProperty("Next", bResult)

    ' Set the initial value of the Shared Property to
    ' 0 if the Shared Property didn't already exist.
    ' This is not entirely necessary but demonstrates
    ' how to initialize a value.
    If bResult = False Then
        spmPropNextReceipt.Value = 0
    End If

    ' Get the next receipt number and update property
    spmPropNextReceipt.Value = spmPropNextReceipt.Value + 1

    ' we are finished and happy
    GetObjectContext.SetComplete

    GetNextReceipt = spmPropNextReceipt.Value

    Exit Function

ErrorHandler:
    GetObjectContext.SetAbort          ' we are unhappy

    ' indicate that an error occured
    GetNextReceipt = -1

    Err.Raise Err.Number, _
        "Bank.GetReceipt.GetNextReceipt", _
        Err.Description

End Function
```

Perform Method, Step5 (Visual Basic)

```
Public Function Perform(ByVal lngPrimeAccount As Long,_
    ByVal lngSecondAccount As Long, ByVal lngAmount _
    As Long, ByVal lngTranType As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' create the account object using our context
    Dim objAccount As Bank.Account
    Set objAccount = _
        GetObjectContext.CreateInstance("Bank.Account")

    If objAccount Is Nothing Then
        Err.Raise ERROR_NUMBER, _
        Description:="Could not create account object"
    End If

    ' call the post function based on the
    ' transaction type
    Select Case lngTranType

        Case 1
            strResult = objAccount.Post(lngPrimeAccount, 0 - lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 2
            strResult = objAccount.Post(lngPrimeAccount, lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 3
            Dim strResult1 As String, strResult2 As String
            ' do the credit
            strResult1 = objAccount.Post(lngSecondAccount, lngAmount)
            If strResult1 = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult1
            Else
                ' then do the debit
                strResult2 = objAccount.Post(lngPrimeAccount, 0 -
lngAmount)
                If strResult2 = "" Then
                ' debit failed
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult2
                Else
                    strResult = strResult1 + "  " + strResult2
                End If
            End If
```

```
        Case Else
            Err.Raise ERROR_NUMBER, _
                Description:="Invalid Transaction Type"

    End Select

    ' Get Receipt Number for the transaction
    Dim objReceiptNo As Bank.GetReceipt
    Dim lngReceiptNo As Long

    Set objReceiptNo = GetObjectContext.CreateInstance("Bank.GetReceipt")
    lngReceiptNo = objReceiptNo.GetNextReceipt
    If lngReceiptNo > 0 Then
        strResult = strResult & "; Receipt No: " _
            & Str$(lngReceiptNo)
    End If

    ' we are finished and happy
    GetObjectContext.SetComplete

    Perform = strResult

    Exit Function

ErrorHandler:

    GetObjectContext.SetAbort          ' we are unhappy

    Perform = ""       ' indicate that an error occured

    Err.Raise Err.Number, "Bank.MoveMoney.Perform", _
        Err.Description

End Function
```

StatefulPerform Method, Step6 (Visual Basic)

```
Public PrimeAccount As Long
Public SecondAccount As Long

Public Function StatefulPerform(ByVal lngAmount _
    As Long, ByVal lngTranType As Long) As String
    StatefulPerform = Perform(PrimeAccount, _
        SecondAccount, lngAmount, lngTranType)
End Function
```

Update Method, Step7 (Visual Basic)

```
Public Function Update() As Long

    On Error GoTo ErrorHandler

    ' get result set and then update table
    ' with new receipt number
    Dim adoConn As New ADODB.Connection
    Dim adoRsReceipt As ADODB.Recordset
    Dim lngNextReceipt As Long
    Dim strSQL As String

    strSQL = "Update Receipt set NextReceipt = NextReceipt + 100"

    adoConn.Open strConnect

    ' Assume that if there is an ado error then
    ' the receipt table does not exist
    On Error GoTo ErrorCreateTable

TryAgain:

    adoConn.Execute strSQL

    strSQL = "Select NextReceipt from Receipt"
    Set adoRsReceipt = adoConn.Execute(strSQL)
    lngNextReceipt = adoRsReceipt!NextReceipt

    Set adoConn = Nothing
    Set adoRsReceipt = Nothing

    ' we are finished and happy
    GetObjectContext.SetComplete

    Update = lngNextReceipt

    Exit Function

ErrorCreateTable:

    On Error GoTo ErrorHandler

    ' create the receipt table
    Dim objCreateTable As CreateTable
    Set objCreateTable = CreateObject("Bank.CreateTable")
    objCreateTable.CreateReceipt

    GoTo TryAgain

ErrorHandler:

    If Not adoConn Is Nothing Then
        Set adoConn = Nothing
    End If
    If Not adoRsReceipt Is Nothing Then
        Set adoRsReceipt = Nothing
```

```
    End If

    GetObjectContext.SetAbort      ' we are unhappy

    Update = -1        ' indicate that an error occured

    Err.Raise Err.Number, "Bank.UpdateReceipt.Update", Err.Description

End Function
```

GetNextReceipt Method, Step7 (Visual Basic)

```
Public Function GetNextReceipt() As Long

    On Error GoTo ErrorHandler

    ' If Shared property does not already exist
    ' it will be initialized
    Dim spmMgr As SharedPropertyGroupManager
    Set spmMgr = CreateObject("MTxSpm.SharedPropertyGroupManager.1")

    Dim spmGroup As SharedPropertyGroup
    Dim bResult As Boolean
    Set spmGroup = _
        spmMgr.CreatePropertyGroup("Receipt", _
        LockMethod, Process, bResult)

    Dim spmPropNextReceipt As SharedProperty
    Set spmPropNextReceipt = _
        spmGroup.CreateProperty("Next", bResult)

    ' Set the initial value of the Shared Property to
    ' 0 if the Shared Property didn’t already exist.
    ' This is not entirely necessary but demonstrates
    ' how to initialize a value.
    If bResult = False Then
        spmPropNextReceipt.Value = 0
    End If

    Dim spmPropMaxNum As SharedProperty
    Set spmPropMaxNum = spmGroup.CreateProperty("MaxNum", bResult)

    Dim objReceiptUpdate As Bank.UpdateReceipt
    If spmPropNextReceipt.Value >= spmPropMaxNum.Value Then
        Set objReceiptUpdate =
GetObjectContext.CreateInstance("Bank.UpdateReceipt")
        spmPropNextReceipt.Value = objReceiptUpdate.Update
        spmPropMaxNum.Value = spmPropNextReceipt.Value + 100
    End If

    ' Get the next receipt number and update property
    spmPropNextReceipt.Value = spmPropNextReceipt.Value + 1

    ' we are finished and happy
    GetObjectContext.SetComplete

    GetNextReceipt = spmPropNextReceipt.Value

    Exit Function

ErrorHandler:
    GetObjectContext.SetAbort          ' we are unhappy

    ' indicate that an error occured
    GetNextReceipt = -1

    Err.Raise Err.Number, "Bank.GetReceipt.GetNextReceipt", Err.Description
```

```
End Function
```

Perform Method, Step8 (Visual Basic)

```
Public Function Perform(ByVal lngPrimeAccount As Long,_
    ByVal lngSecondAccount As Long, ByVal lngAmount _
    As Long, ByVal lngTranType As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' check for security
    If (lngAmount > 500 Or lngAmount < -500) Then
        If Not GetObjectContext.IsCallerInRole("Managers") Then
            Err.Raise Number:=APP_ERROR, _
            Description:="Need 'Managers' role for amounts over $500"
        End If
    End If

    ' create the account object using our context
    Dim objAccount As Bank.Account
    Set objAccount = _
        GetObjectContext.CreateInstance("Bank.Account")

    If objAccount Is Nothing Then
        Err.Raise ERROR_NUMBER, _
        Description:="Could not create account object"
    End If

    ' call the post function based on the
    ' transaction type
    Select Case lngTranType

        Case 1
            strResult = objAccount.Post(lngPrimeAccount, 0 - lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 2
            strResult = objAccount.Post(lngPrimeAccount, lngAmount)
            If strResult = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult
            End If

        Case 3
            Dim strResult1 As String, strResult2 As String
            ' do the credit
            strResult1 = objAccount.Post(lngSecondAccount, lngAmount)
            If strResult1 = "" Then
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult1
            Else
                ' then do the debit
                strResult2 = objAccount.Post(lngPrimeAccount, 0 -
lngAmount)
```

```
                If strResult2 = "" Then
                ' debit failed
                Err.Raise ERROR_NUMBER, _
                    Description:=strResult2
                Else
                    strResult = strResult1 + "  " + strResult2
                End If
            End If

        Case Else
            Err.Raise ERROR_NUMBER, _
                Description:="Invalid Transaction Type"

    End Select

    ' Get Receipt Number for the transaction
    Dim objReceiptNo As Bank.GetReceipt
    Dim lngReceiptNo As Long

    Set objReceiptNo = GetObjectContext.CreateInstance("Bank.GetReceipt")
    lngReceiptNo = objReceiptNo.GetNextReceipt
    If lngReceiptNo > 0 Then
        strResult = strResult & "; Receipt No: " _
            & Str$(lngReceiptNo)
    End If

    ' we are finished and happy
    GetObjectContext.SetComplete

    Perform = strResult

    Exit Function

ErrorHandler:

    GetObjectContext.SetAbort          ' we are unhappy

    Perform = ""       ' indicate that an error occured

    Err.Raise Err.Number, "Bank.MoveMoney.Perform", _
        Err.Description

End Function
```

Post Method, Step8 (Visual Basic)

```
Public Function Post(ByVal lngAccountNo As Long, _
    ByVal lngAmount As Long) As String

    Dim strResult As String

    On Error GoTo ErrorHandler

    ' check for security
    If (lngAmount > 500 Or lngAmount < -500) Then
        If Not GetObjectContext.IsCallerInRole("Managers") Then
            Err.Raise Number:=APP_ERROR, _
            Description:="Need 'Managers' role for amounts over $500"
        End If
    End If

    ' obtain the ADO environment and connection
    Dim adoConn As New ADODB.Connection
    Dim varRows As Variant

    adoConn.Open strConnect

    On Error GoTo ErrorCreateTable

    ' update the balance
    Dim strSQL As String
    strSQL = "UPDATE Account SET Balance = Balance + "_
        + Str$(lngAmount) + " WHERE AccountNo = "
        + Str$(lngAccountNo)

TryAgain:
    adoConn.Execute strSQL, varRows

    ' if anything else happens
    On Error GoTo ErrorHandler

    ' get resulting balance which may have been
    ' further updated via triggers
    strSQL = "SELECT Balance FROM Account " _
        + "WHERE AccountNo = " + Str$(lngAccountNo)

    Dim adoRS As ADODB.Recordset
    Set adoRS = adoConn.Execute(strSQL)
    If adoRS.EOF Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
            + Str$(lngAccountNo) + " not on file."
    End If

    Dim lngBalance As Long
    lngBalance = adoRS.Fields("Balance").Value

    ' check if account is overdrawn
    If (lngBalance) < 0 Then
        Err.Raise Number:=APP_ERROR, _
            Description:="Error. Account " _
```

```
            + Str$(lngAccountNo) _
            + " would be overdrawn by " _
            + Str$(lngBalance) + ". Balance is still "
            + Str$(lngBalance - lngAmount) + "."
    Else
        If lngAmount < 0 Then
            strResult = strResult _
                & "Debit from account "
                & lngAccountNo & ", "
        Else
            strResult = strResult _
                & "Credit to account "
                & lngAccountNo & ", "
        End If
        strResult = strResult + "balance is $"
            & Str$(lngBalance) & ". (VB)"
    End If

    ' cleanup
    Set adoRS = Nothing
    Set adoConn = Nothing

    ' we are finished and happy
    GetObjectContext.SetComplete

    Post = strResult

Exit Function

ErrorCreateTable:
    On Error GoTo ErrorHandler

    ' create the account table
    Dim objCreateTable As CreateTable
    Set objCreateTable = _
    GetObjectContext.CreateInstance("Bank.CreateTable")
    objCreateTable.CreateAccount

    GoTo TryAgain

ErrorHandler:
    ' cleanup
    If Not adoRS Is Nothing Then
        Set adoRS = Nothing
    End If
    If Not adoConn Is Nothing Then
        Set adoConn = Nothing
    End If

    GetObjectContext.SetAbort          ' we are unhappy

    Post = ""         ' indicate that an error occurred
    Err.Raise Err.Number, "Bank.Accout.Post", _
        Err.Description

End Function
```

Utwórz nowy obiekt

Pozwala dodaæ nowe elementy do okna programu Microsoft Transaction Server Explorer. Skutek wykonania polecenia **Nowy** zale¿y od tego, jaki folder jest aktualnie przegl¹dany w prawym okienku programu. Na przyk³ad jeœli otwarty jest folder "Komputery", klikniêcie polecenia **Nowy** spowoduje dodanie komputera.

Skrót paska narzêdzi:

Informacje na temat dodawania komputerów do foldera "Komputery" mo¿na znaleŸæ pod has³em Konfiguracja Ÿród³owego serwera rozmieszczania.

Informacje na temat tworzenia nowych pakietów mo¿na znaleŸæ pod has³em Tworzenie pustego pakietu MTS.

Informacje na temat tworzenia nowych ról mo¿na znaleŸæ pod has³em Dodawanie nowej roli.

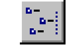
Du¿e ikony

Powoduje, ¿e widoczne w prawym okienku programu elementy hierarchii s¹ wyœwietlane w du¿ym formacie.

Skrót paska narzêdzi:

Ma³e ikony

Powoduje, ¿e widoczne w prawym okienku programu elementy hierarchii s¹ wyœwietlane w ma³ym formacie.

Skrót paska narzêdzi: 

Lista

Powoduje, ¿e widoczne w prawym okienku programu elementy hierarchii s¹ wyœwietlane w postaci listy.

Skrót paska narzêdzi:

W³aœciwoœci

Pozwala wyœwietliæ ustawienia w³aœciwoœci elementów znajduj¹cych siê w zaznaczonym aktualnie folderze. W³aœciwoœci s¹ wyœwietlane w prawym okienku programu Microsoft Transaction Server Explorer, w formacie kolumnowym.

Skrót paska narzêdzi:

Pokazywane w³aœciwoœci s¹ zale¿ne od typu wybranego obiektu. W poni¿szej tabeli zestawiono w³aœciwoœci wyœwietlane dla ró¿nych folderów programu.

Folder	W³aœciwoœci
"Komputery"	**Nazwa** Nazwa przypisana komputerowi i rozpoznawana w ramach domeny systemu Windows NT. Komputer, na którym dzia³a program Transaction Server, jest rozpoznawany pod nazw¹ "Mój Komputer".
	Limit czasu
"Zainstalowane pakiety"	**Nazwa** Nazwa przypisana pakietowi.
	Zabezpieczenia Pokazuje, czy dla pakietu w³¹czono zabezpieczenia.
	Uwierzytelnienia Poziom sprawdzania uwierzytelnieñ.
	Zamkniêcie Czas, po jakim pakiet jest zamykany w razie nieaktywnoœci.
	Uruchamiany zawsze Powoduje, ¿e w razie nieatywnoœci pakiet nie jest zamykany .
	Konto Konto systemu Windows NT, ustawione jako to¿samoœæ pakietu.
	ID pakietu Skojarzony z pakietem uniwersalny unikatowy identyfikator.
"Sk³adniki"	**Prog ID** Nazwa skojarzona ze sk³adnikiem.
	Transakcja Pokazuje, czy sk³adnik obs³uguje transakcje.
	DLL
	CLSID
	U¿yj MTX Pokazuje, czy sk³adnik jest uruchomiony w œrodowisku programu Transaction Server.
	W procesie Wskazuje na fakt, ¿e sk³adnik jest uruchomiony w procesie innego serwera.
	Lokalny Wskazuje na fakt, ¿e sk³adnik jest uruchomiony w procesie serwera na komputerze lokalnym.
	Zdalny Wskazuje na fakt, ¿e sk³adnik jest uruchomiony w procesie serwera na komputerze zdalnym.
	Serwer Nazwa komputera zdalnego, na którym jest uruchomiony sk³adnik.
	W¹tkowoœæ Model w¹tków sk³adnika.

	Zabezpieczenia Pokazuje, czy dla sk³adnika w³¹czono zabezpieczenia.
"Role"	**Nazwa** Nazwa przypisana roli.
	ID roli Skojarzony z rol¹ unikatowy identyfikator.
"Interfejsy"	**Nazwa** Nazwa przypisana do interfejsu.
	ID interfejsu Skojarzony z interfejsem unikatowy identyfikator.
	Proxy DLL
	Plik biblioteki typów Nazwa pliku zawieraj¹cego bibliotekê typów.
"Metody"	**Nazwa** Nazwa przypisana metodzie.
"Przynale¿noœæ ról"	**Nazwa** Nazwa przypisana roli, któr¹ dodano do sk³adnika lub interfejsu.
	ID roli Skojarzony z rol¹ unikatowy identyfikator.

Status

Pozwala obejrzeæ bie¿¹cy stan sk³adnika lub komputera.

Skrót paska narzêdzi:

Status komputera

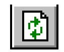
- **Nazwa** Nazwa skojarzona z komputerem.
- **DTC** Pokazuje, czy jest uruchomiona us³uga MS DTC.

Status sk³adnika

- **Prog ID** Nazwa skojarzona ze sk³adnikiem.
- **Obiekty** Ca³kowita liczba obiektów alokowanych w ramach procesu serwera.
- **Uaktywnione** Ca³kowita liczba obiektów, z których korzystaj¹ klienci.
- **W wywo³aniu** Ca³kowita liczba obiektów, które aktualnie s¹ wywo³ywane przez klienta.

Odœwie¿, polecenie (menu Widok)

Pozwala rêcznie zaktualizowaæ informacje wyœwietlane w prawym okienku programu Microsoft Transaction Server Explorer.

Skrót paska narzêdzi: 

Słownik

ACID
ActiveX
administrator
agregacja
aktywacja na żądanie
atomowość
automatyczna transakcja
bezpieczne odwołanie
biblioteka dołączana dynamicznie (DLL)
biblioteka Resource Dispenser Manager
biblioteka typów
bezpośredni twórca
bezpośredni wywołujący
domena
dziedzina aktywności
grupa
identyfikator klasy (CLSID)
identyfikator programistyczny (progID)
identyfikator zabezpieczeń (SID)
instancja
interfejs
interfejs ODBC (z ang. Open Database Connectivity)
izolacja
izolacja błędów
izolacja procesów
katalog
klasa
klaster
klient
klient bazowy
klient/serwer
konstruktor
konstruktor klasy
kontekst
kontekst transakcji
konto globalne
konto lokalne

komunikat śledzenia
limit czasu transakcji
menedżer transakcji
menedżer zasobów
metoda
Microsoft Distributed Transaction Coordinator (MS DTC)
Microsoft Transaction Server Explorer
model DCOM
narzędzie do tworznia plików wykonywalnych aplikacji
nazwa użytkownika
nazwa źródła danych (DSN)
Null
obiekt
obiekt nie zachowujący stanu
obiekt programu Microsoft Transaction Server
obiekt zachowujący stan
OLE Transactions
oryginalny twórca
oryginalny wywołujący
pakiet
pakiet biblioteki
pakiet serwera
pakiet wstępnie wbudowany
plik pakietu
proces bazowy
proces serwera
protokół XA
proxy
przekaz dwufazowy
przekazywanie międzyprocesowe
przystawka
przyśpieszone wykrywanie błędów
replikacja
rola
rozdzielacz zasobów
rozdzielacz zasobów ODBC
równoważenie obciążeń
semafor
składnik
składnik programu Microsoft Transaction Server

składnik wewnątrzprocesowy

składnik zewnątrzprocesowy

składnikowy model obiektów COM

stub

tolerancja błędów

tożsamość

transakcja

transakcja wątpliwa

trwałość

twórca

umieszczanie w pulach

uwierzytelnianie

wartość logiczna

wątek

wątek główny

wątek szufladkowy

właściwości obiektu kontekstowego

właściwość współużytkowana

współbieżność

wyjątek

wyrażenie typu string

wywołanie procedury zdalnej (RPC)

wywołujący

zabezpieczenia deklaratywne

zabezpieczenia programistyczne

zakleszczenie

zalogowany użytkownik interakcyjny

zasada działań

zdalny składnik

zmienna obiektowa

zwartość

ACID

Skrót od czterech słów angielskich atomicity, consistency, isolation i durability wyrażających podstawowe własności transakcji: atomowość, zwartość, izolację i trwałość.

ActiveX

Zbiór technologii umożliwiający wzajemną współpracę składników oprogramowania w środowisku sieciowym, niezależnie od języka programowania, w jakim je utworzono. Zbiór technologii ActiveX stanowi element składnikowego modelu obiektów COM (z ang. Component Object Model).

administrator

Użytkownik korzystający z programu Microsoft Transaction Server Explorer w celu instalowania, konfiguruwania

oraz zarządzania składnikami i pakietami programu Microsoft Transaction Server.

agregacja

Technika implementacji obiektów składnikowych, zgodnie z którą nowe obiekty mogą być budowane za pomocą jednego lub większej liczby istniejących obiektów obsługujących kilka lub wszystkie z wymaganych interfejsów nowego obiektu.

aktywacja na żądanie

Możliwość uaktywniania obiektów programu Microsoft Transaction Server tylko w razie konieczności, w odpowiedzi na żądanie klienta. Obiekty mogą być dezaktywowane nawet, jeśli klient nie zwolnił odwołań do nich, dzięki czemu nieużywane zasoby serwera mogą być wykorzystywane bardziej efektywnie.

atomowość

Cecha transakcji, wskazująca, że wykonywane są albo wszystkie operacje transakcji, albo nie jest wykonywana żadna.

automatyczna transakcja

Transakcja tworzona w środowisku czasu wykonywania programu Microsoft Transaction Server na podstawie atrybutu transakcji składnika.

bezpieczne odwołanie

Odwołanie do obiektu bieżącego, któremu nie grozi przekazanie na zewnątrz kontekstu obiektu bieżącego.

biblioteka dołączana dynamicznie (DLL)

Plik zawierający jedną lub więcej funkcji, które są kompilowane, dołączane i przechowywane oddzielnie w stosunku do wykorzystujących je procesów. Kiedy dany proces jest rozpoczynany lub uruchamiany, system operacyjny mapuje biblioteki DLL do jego przestrzeni adresowej.

biblioteka Resource Dispenser Manager

Biblioteka dołączana dynamicznie (DLL), która koordynuje prace w ramach kolekcji rozdzielaczy zasobów.

biblioteka typów

Plik zawierający standardowe opisy typów danych, modułów i interfejsów, które mogą być przydatne w pełnym wykorzystaniu mechanizmów technologii ActiveX.

bezpośredni twórca

Tożsamość procesu (klienta bazowego lub procesu serwera), który bezpośrednio utworzył bieżący obiekt.

bezpośredni wywołujący

Tożsamość procesu (klienta bazowego lub procesu serwera) wywołującego obiekty w ramach bieżącego procesu serwera.

domena

W systemie Windows NT jest to kolekcja komputerów zdefiniowania przez administratora sieci serwera systemu Windows NT, współużytkująca wspólną bazę danych katalogów. Domena zapewnia dostęp do scentralizowanych kont użytkowników i grup, zarządzanych przez administratora domeny. Każda domena ma unikatową nazwę.

dziedzina aktywności

Kolekcja obiektów programu Microsoft Transaction Server o pojedynczym, rozpowszechnianym, logicznym wątku wykonywania. Każdy obiekt programu Microsoft Transaction Server przynależy do jednej dziedziny aktywności.

grupa

Nazwa identyfikująca zbiór składający się z jednego lub kilku kont użytkowników systemu Windows NT.

identyfikator klasy (CLSID)

Uniwersalny unikatowy identyfikator (UUID), który identyfikuje składnik COM. Identyfikatory CLSID poszczególnych składników COM są przechowywane w rejestrze systemu Windows, dzięki czemu składniki te mogą być pobierane z różnych aplikacji.

identyfikator programistyczny (progID)

Nazwa identyfikująca składnik COM. Na przykład identyfikatorem programistycznym mogłaby być nazwa Bank.MoveMoney.

identyfikator zabezpieczeń (SID)

Unikatowa nazwa identyfikująca zalogowanego użytkownika w systemie zabezpieczeń. Identyfikator SID może odnosić się do pojedynczego użytkownika lub grupy użytkowników.

instancja

Obiekt konkretnej klasy składnika. Każda instancja ma swoje własne, prywatne dane oraz zmienne. Instancja składnika jest synonimem pojęcia obiekt.

interfejs

Grupa logicznie powiązanych operacji i metod, zapewniających dostęp do obiektów składnika.

interfejs ODBC (z ang. Open Database Connectivity)

Standardowy interfejs języków programowania używany do łączenia z różnymi źródłami danych.

izolacja

Stan, w którym wydaje się, że dwie transakcje uruchomione współbieżnie działają w izolacji. Użytkownik ma wrażenie, że system uruchamia transakcje kolejno (po jednej).

izolacja błędów

Mechanizm powodujący, że błędy lub awarie mające miejsce w ramach danego składnika nie są przenoszone do innych składników systemu, lecz oddziaływują jedynie na ten składnik.

izolacja procesów

Technika uruchamiania procesu serwera w oddzielnym obszarze pamięci tak, aby odizolować ten proces od innych procesów serwera. Dzięki izolacji proces serwera jest chroniony przed błędami krytycznymi, które mogłyby wystąpić w innych procesach aplikacji. Mechanizm izolacji pozwala ponadto zapobiec zakończeniu innego procesu serwera w razie wystąpienia krytycznego błędu aplikacji. Pakiety MTS, które obsługują mechanizm izolowania procesów są nazywane pakietami serwera.

katalog

Magazyn danych programu Microsoft Transaction Server, w którym są przechowywane informacje o konfiguracji składników, pakietów i ról. Do administrowania katalogiem służy program Microsoft Transaction Server Explorer.

klasa

Typ definiujący interfejsy obiektów konkretnego typu. Klasa definiuje właściwości obiektu oraz metody sterowania jego zachowaniem.

klaster

Dwa lub więcej niezależnych systemów komputerowych, które za pośrednictwem programu Microsoft Cluster

Server mogą być traktowane i obsługiwane jak pojedynczy system.

klient

Aplikacja lub proces korzystający z usługi innego procesu lub składnika.

klient bazowy

Klient uruchomiony na zewnątrz środowiska czasu wykonywania programu Microsoft Transaction Server, ale uprawniony do tworzenia instancji obiektów programu.

klient/serwer

Model aplikacji rozproszonych, w którym aplikacje klienckie mogą żądać usług od aplikacji serwerowych. Serwer może obsługiwać wielu klientów jednocześnie, a klient może żądać danych z wielu serwerów. Dana aplikacja może być jednocześnie kliencka i serwerowa.

konstruktor

Specjalna funkcja inicjująca, używana w językach programowania C++ i Java, wywoływana za każdym razem, kiedy jest deklarowana nowa instancja klasy. Funkcja ta zapobiega błędom powstającym podczas używania obiektów niezainicjowanych. Konstruktor ma tę samą nazwę co klasa, ale zwraca wartość.

konstruktor klasy

Obiekt implementujący interfejs **IClassFactory**, który z kolei pozwala mu tworzyć obiekty konkretnej klasy.

kontekst

Stan skojarzony domyślnie z danym obiektem programu Microsoft Transaction Server. Kontekst zawiera informacje o o środowisku wykonywania obiektu takie, jak tożsamość twórcy obiektu i, opcjonalnie, transakcja towarzysząca działaniom obiektu. Pojęcie to ma wiele cech wspólnych z kontekstem procesu, który występuje na poziomie systemu operacyjnego i jest związany z wykonywaniem konkretnego programu. W środowisku czasu wykonywania programu Microsoft Transaction Server są obsługiwane konteksty wszystkich obiektów.

kontekst transakcji

Obiekt pozwalający klientowi dynamicznie dołączać do jednej transakcji jeden lub większą liczbę obiektów.

konto globalne

Zwykłe konto użytkownika w jego domenie macierzystej. Większość kont są to konta globalne, co jest ustawieniem domyślnym. Jeśli w sieci występuje wiele domen, najlepiej jeśli każdy użytkownik sieci dysponuje tylko jednym kontem globalnym w jednej domenie.

konto lokalne

Konto przydzielane w lokalnej domenie użytkownikowi, którego zwykłe konto nie znajduje się w zaufanej domenie. Konta lokalne nie mogą być używane do logowania interakcyjnego. Konta lokalne utworzone w danej domenie nie mogą być używane w zaufanych domenach.

komunikat śledzenia

Komunikat informujący o bieżącym stanie różnych działań programu Microsoft Transaction Server takich, jak uruchamianie czy zamknięcie.

limit czasu transakcji

Maksymalny czas aktywności transakcji, po którym jest ona automatycznie przerywana przez menedżera transakcji.

menedżer transakcji

Usługa systemowa odpowiedzialna za koordynowanie wyników transakcji tak, aby zapewnić ich atomowość. Menedżer transakcji pozwala menedżerom zasobów podejmować zgodne decyzje co do przerwania lub przekazania transakcji.

menedżer zasobów

Usługa systemu zarządzająca danymi trwałymi. Aplikacje serwera korzystają z menedżerów zasobów w celu utrzymania stabilnego stanu aplikacji, na przykład zapisu inwentaryzacji, kolejności oczekujących zadań czy dostępnych kont. Menedżer zasobów współpracuje z menedżerem transakcji, dzięki czemu można zapewnić aplikacjom izolację i atomowość (za pomocą protokołu przekazywania dwufazowego). Przykładem menedżera zasobów jest program Microsoft SQL Server.

metoda

Procedura (funkcja) wykonywana przez obiekt.

Microsoft Distributed Transaction Coordinator (MS DTC)

Usługa zarządzająca transakcjami, która koordynuje transakcje obejmujące wiele menedżerów zasobów. Całość działań może być przekazywana jako transakcja atomowa, nawet jeśli obejmuje ona wiele menedżerów zasobów (potencjalnie na oddzielnych komputerach).

Microsoft Transaction Server Explorer

Aplikacja służąca do konfigurowania i zarządzania składnikami programu Microsoft Transaction Server w rozproszonej sieci komputerowej.

model DCOM

Model DCOM jest protokołem obiektowym umożliwiającym wzajemną, bezpośrednią komunikację składników ActiveX za pośrednictwem sieci. Model DCOM jest niezależny od języków programowania, dzięki czemu do tworzenia aplikacji DCOM można wykorzystać dowolny język pozwalający programować składniki ActiveX .

narzędzie do tworzenia plików wykonywalnych aplikacji

Funkcja programu MTS Explorer, pozwalająca tworzyć pliki wykonywalne aplikacji poprzez eksport pakietów.

nazwa użytkownika

Nazwa identyfikująca dane konto użytkownika systemu Windows NT.

nazwa źródła danych (DSN)

Skrót od angielskiego określenia "Data Source Name". Nazwa używana przez aplikacje w celu uzyskania połączenia ze źródłem danych ODBC.

Null

Wartość, która oznacza dane nieznane lub brakujące.

obiekt

Instancja czasu wykonywania składnika COM. Obiekty są tworzone w klasach składnika. Obiekt stanowi synonim pojęcia instancja.

obiekt nie zachowujący stanu

Obiekt nie utrzymujący indywidualnego stanu będącego wynikiem obsługi jednego lub większej liczby wywołań klientów.

obiekt programu Microsoft Transaction Server

Obiekt typu COM, zgodny z modelem programowania i rozmieszczania programu Microsoft Transaction Server,

wykonywany w środowisku czasu wykonywania tego programu.

obiekt zachowujący stan

Obiekt utrzymujący indywidualny stan będący wynikiem obsługi jednego lub większej liczby wywołań klientów.

OLE Transactions

OLE Transactions jest obiektowym protokołem przekazywania dwufazowego, opartym na modelu COM. Protokół ten jest używany przez menedżerów zasobów w celu zapewnienia uczestnictwa w transakcjach rozpowszechnianych, koordynowanych przez usługę Microsoft Distributed Transaction Coordinator (DTC).

oryginalny twórca

Tożsamość klienta bazowego, który utworzył bieżący obiekt. Oryginalny twórca oraz oryginalny wywołujący nie pokrywają się tylko wówczas, gdy oryginalny twórca przesłał obiekt do innego klienta bazowego. Zobacz też: oryginalny wywołujący.

oryginalny wywołujący

Tożsamość klienta bazowego, który zainicjował działania.

pakiet

Zbiór składników aplikacji wykonujących związane ze sobą funkcje. Wszystkie składniki pakietu są uruchamiane razem, w tym samym procesie serwera programu Microsoft Transaction Server. Dany pakiet wyznacza granice zaufania, w ramach których są sprawdzane uwierzytelnienia, a tworzące go składniki są rozmieszczane jako całość. Do tworzenia pakietów służy program Transaction Server Explorer. Można wyróżnić dwa rodzaje pakietów: pakiety bibliotek oraz pakiety serwera.

pakiet biblioteki

Pakiet działający w procesie klienta, który go utworzył. Pakiety bibliotek nie obsługują śledzenia składników, sprawdzania ról oraz izolowania procesów. W programie MTS występują dwa rodzaje pakietów : pakiety bibliotek i pakiety serwera.

pakiet serwera

Pakiet uruchamiany osobno, w swoim własnym procesie na komputerze lokalnym. Pakiety serwera obsługują zabezpieczenia oparte na rolach, współużytkowanie zasobów, izolowanie procesów oraz zarządzanie procesami (na przykład śledzenie pakietów). Program MTS obsługuje dwa rodzaje pakietów: pakiety bibliotek i pakiety serwera.

pakiet wstępnie wbudowany

Plik pakietu, który zawiera informacje o składnikach i rolach pakietu. Plik pakietu jest tworzony za pomocą funkcji eksportu pakietów programu Transaction Server Explorer. Kiedy jest tworzony pakiet wstępnie wbudowany, skojarzone z nim pliki składników (biblioteki DLL, biblioteki typów i ewentualnie biblioteki DLL typu proxy-stub) są kopiowane do tego samego katalogu, w którym utworzono plik pakietu.

plik pakietu

Plik zawierający informacje o składnikach i rolach pakietu. Plik pakietu jest tworzony za pomocą funkcji eksportu pakietów programu Transaction Server Explorer. Kiedy jest tworzony pakiet wstępnie wbudowany, skojarzone z nim pliki składników (biblioteki DLL, biblioteki typów i ewentualnie biblioteki DLL typu proxy-stub) są kopiowane do tego samego katalogu, w którym utworzono plik pakietu.

proces bazowy

Proces aplikacji, w którym jest wykonywany klient bazowy. Klient bazowy działa na zewnątrz środowiska czasu wykonywania programu Microsoft Transaction Server, ale może tworzyć instancje obiektów programu.

proces serwera

Proces macierzysty składników programu Microsoft Transaction Server.

Składnik programu Microsoft Transaction Server może być pobrany w zastępczym procesie serwera albo na komputerze klienta, albo w procesie aplikacji klienckiej.

protokół XA

Protokół przekazu dwufazowego zdefiniowany przez grupę X/Open DTP. Protokół XA jest obsługiwany przez wiele baz danych systemu Unix, między innymi bazy danych Informix, Oracle i DB2.

proxy

Obiekt związany z konkretnym interfejsem, zapewniający międzyprocesowe przekazywanie parametrów oraz komunikację niezbędne, aby klient mógł wywołać obiekt aplikacji uruchomiony w innym środowisku wykonywania, na przykład w innym wątku lub procesie. Obiekt proxy jest zlokalizowany po stronie klienta i komunikuje się z odpowiadającym mu obiektem stub, który jest zlokalizowany po stronie wywoływanego obiektu aplikacji.

przekaz dwufazowy

Sposób przekazu, który zapewnia, że transakcje stosowane do więcej niż jednego serwera zostaną zakończone na wszystkich serwerach lub nie zostaną przeprowadzone w ogóle. Przekaz dwufazowy jest koordynowany przez menedżera transakcji i obsługiwany przez menedżerów zasobów.

przekazywanie międzyprocesowe

Proces grupowania w pakiety i przesyłania parametrów metod interfejsu przez granice wątków lub procesów.

przystawka

Program administracyjny zarządzany przez program Microsoft Management Console (MMC). Program MTS Explorer dla systemu Windows NT jest przystawką.

przyśpieszone wykrywanie błędów

Funkcja programu Microsoft Transaction Server ułatwiająca wykrywanie błędów i zapobieganie ich skutkom. Kiedy program Transaction Server wykryje nieoczekiwane warunki, które mogłyby spowodować błąd wewnętrzny, natychmiast kończy proces i zapisuje w dzienniku zdarzeń systemu Windows NT szczegółowy komunikat o awarii.

replikacja

Operacja kopiowania katalogu z pamięci jednego komputera do pamięci innego. Replikację wykorzystuje się do synchronizacji sklastrowanych serwerów MTS.

rola

Symboliczna nazwa definiująca klasę użytkowników związanych z określonym zbiorem składników. Każda z ról wyznacza zbiór użytkowników uprawnionych do wywoływania interfejsów danego składnika.

rozdzielacz zasobów

Usługa zapewniająca w ramach procesu synchronizację nietrwałych zasobów i zarządzanie nimi, dzięki czemu obiekty programu Microsoft Transaction Server mogą w prosty i efektywny sposób współużytkować zasoby. Na przykład rozdzielacz zasobów ODBC zarządza pulami połączeń z bazą danych.

rozdzielacz zasobów ODBC

Rozdzielacz zasobów obsługujący pule połączeń bazy danych dla składników programu Microsoft Transaction Server, które korzystają ze standardowych interfejsów programistycznych ODBC.

równoważenie obciążeń

Zdolność równomiernego rozkładania zadań pomiędzy kilkoma serwerami w sieci, dzięki której wzrasta ogólna

wydajność sieci.

semafor

Mechanizm blokujący, stosowany wewnątrz menedżerów zasobów i rozdzielaczy zasobów. Semafory nie mają nazw symbolicznych, zapewniają dostęp tylko w trybie wyłączności i współużytkowania, nie wykrywają zakleszczeń oraz nie mogą być automatycznie zwalniane ani podnoszone.

składnik

Kod programu oparty na technologii ActiveX, traktowany jako całość, dostarczający zbioru konkretnych usług za pośrednictwem wyspecjalizowanych interfejsów. Składniki dostarczają obiektów, których klienci mogą żądać w czasie wykonywania.

składnik programu Microsoft Transaction Server

Składnik COM wykonywany w środowisku czasu wykonywania programu Microsoft Transaction Server. Składnik programu Transaction Server musi być biblioteką dołączaną dynamicznie (DLL), która implementuje klasę pozwalającą tworzyć obiekty oraz opisuje wszystkie interfejsy składnika (w bibliotece typów do standardowego przekazywania).

składnik wewnątrzprocesowy

Składnik uruchamiany w przestrzeni procesu klienta. Zazwyczaj jest to biblioteka dołączana dynamicznie (DLL).

składnik zewnątrzprocesowy

Składnik uruchamiany w oddzielnej przestrzeni procesowej swojego klienta. Nawet jeśli składniki są implementowane jako biblioteki DLL, program Microsoft Transaction Server pozwala uruchomić je na zewnątrz klienta poprzez pobranie ich w zastępczych procesach serwera.

składnikowy model obiektów COM

Otwarta architektura międzysystemowego projektowania aplikacji typu klient/serwer, oparta na technologii obiektów. Klienci mogą uzyskiwać dostęp do obiektu za pośrednictwem interfejsu zaimplementowanego dla danego obiektu. Model COM jest niezależny od języków programowania, dzięki czemu aplikacje COM można programować w dowolnym języku pozwalającym tworzyć składniki ActiveX. Określenie COM jest skrótem wyrażenia angielskiego "Component Object Model".

stub

Obiekt związany z konkretnym interfejsem, zapewniający międzyprocesowe przekazywanie parametrów oraz komunikację niezbędne, aby obiekt aplikacji mógł odbierać wywołania od klientów uruchomionych w innym środowisku wykonywania, na przykład w innym wątku lub procesie. Obiekt stub jest zlokalizowany po stronie obiektu aplikacji i komunikuje się z odpowiadającym mu obiektem proxy, który jest zlokalizowany po stronie wywołującego go klienta.

tolerancja błędów

Zdolność systemu do odtworzenia własnego stanu po wystąpieniu błędu, awarii lub zmianie warunków środowiskowych (na przykład po przerwie zasilania). Prawdziwa tolerancja, w odróżnieniu od tolerancji ręcznej, która polega na przywracaniu danych ze sporządzonych wcześniej kopii zapasowych, zapewnia w pełni automatyczne odtworzenie danych bez ingerencji w pliki lub działania użytkownika.

tożsamość

Właściwość pakietu określająca konta użytkowników uprawnione do korzystania z tego pakietu. Może być to albo konto konkretnego użytkownika, albo grupa użytkowników danej domeny systemu Windows NT.

transakcja

Fragment działań realizowany jako operacja atomowa—to znaczy operacja, która jest wykonywana lub nie wykonywana w całości.

transakcja wątpliwa

Transakcja, która została przygotowana, ale nie może zostać ani przerwana, ani przekazana z powodu niedostępności koordynującego ją serwera.

trwałość

Niewrażliwość na awarie.

twórca

Klient tworzący obiekt dostarczany przez składnik (za pomocą metody **CreateObject**, **CoCreateInstance** lub **CreateInstance**). Po utworzeniu obiektu klient uzyskuje odwołania do niego, za pomocą których może wywoływać metody obiektu. Zobacz też: wywołujący.

umieszczanie w pulach

Funkcja optymalizacji wydajności, wykorzystująca kolekcje wstępnie alokowanych zasobów takich, jak obiekty i połączenia z bazą danych. Umieszczanie w pulach pozwala bardziej efektywnie alokować zasoby.

uwierzytelnianie

Proces sprawdzania tożsamości użytkowników usiłujących uzyskać dostęp do systemu. Do uwierzytelniania użytkowników bardzo często stosuje się hasła.

wartość logiczna

Wartość typu prawda/fałsz lub tak/nie.

wątek

Podstawowy obiekt, dla którego system operacyjny alokuje czas procesora. W wątku może być wykonywany dowolny fragment kodu aplikacji, nie wyłączając kodu wykonywanego w danej chwili w innym wątku. Wszystkie wątki danego procesu współużytkują wirtualną przestrzeń adresów, zmienne globalne i zasoby systemu operacyjnego dla procesu.

wątek główny

Pojedynczy wątek, w którym są uruchamiane wszystkie obiekty składników "jednowątkowych". Zobacz też: wątek szufladkowy.

wątek szufladkowy

Wątek, w którym są realizowane wywołania obiektów składników skonfigurowanych jako "szufladkowe". Każdy obiekt "jest umieszczany w oddzielnej szufladce" (wątku) i pozostaje w niej do końca swojego istnienia. Wszystkie wywołania obiektu są realizowane w przypisanej mu szufladce (wątku). Tego rodzaju model wątków jest używany, na przykład, w implementacjach składników, gdzie stan wątku jest przechowywany w lokalnym magazynie wątków TLS (z ang. thread local storage). Obiekty danego składnika mogą być rozmieszczane w wielu szufladkach. Zobacz też: wątek główny.

właściwości obiektu kontekstowego

Właściwości, które można uzyskać od obiektu kontekstowego, na przykład obiekty wewnętrzne programu Internet Information Server.

właściwość współużytkowana

Zmienna, która za pośrednictwem narzędzia Menedżer właściwości współużytkowanych jest udostępniana wszystkim obiektom w ramach tego samego procesu serwera. Właściwość taka może przybierać wartości dowolnego typu, reprezentowane przez typ "variant".

współbieżność

Zdolność wykonywania procesów lub transakcji jednocześnie z innymi zadaniami.

wyjątek

Występujące w czasie wykonywania programu wyjątkowe zdarzenie lub błąd, który wymaga ingerencji oprogramowania uruchamianego poza zwykłym tokiem działań.

wyrażenie typu string

Wyrażenie, którego wynikiem jest ciąg następujących po sobie znaków.

wywołanie procedury zdalnej (RPC)

Standard pozwalający danemu procesowi wywoływać funkcje wykonywane w innym procesie. Ten drugi proces może być realizowany zarówno na tym samym komputerze, jak i na innym komputerze sieci. RPC jest skrótem od angielskiego określenia "Remote Procedure Call".

wywołujący

Klient wywołujący metodę obiektu. Wywołujący nie zawsze musi być twórcą obiektu. Na przykład klient A mógłby utworzyć obiekt X i przekazać odwołanie do niego klientowi B; klient B mógłby wywołać metodę obiektu X za pomocą uzyskanego odwołania. W takim wypadku klient A byłby twórcą obiektu, a klient B wywołującym. Zobacz też: twórca.

zabezpieczenia deklaratywne

Zabezpieczenia konfigurowane za pomocą programu Microsoft Transaction Server Explorer. Dostęp do pakietów, składników i interfejsów można kontrolować definiując role. Role określają, którzy użytkownicy mogą wywoływać interfejsy składnika. Zobacz też: zabezpieczenia programistyczne.

zabezpieczenia programistyczne

Dostarczane przez składnik procedury, pozwalające sprawdzać, czy klient jest uprawniony do wykonania żądanej operacji. Zobacz też: zabezpieczenia deklaratywne.

zakleszczenie

Sytuacja, w której dwa lub więcej wątków zostaje zablokowanych na trwałe (w stanie oczekiwania); każdy z wątków oczekuje na zasób, który może zwolnić jedynie jeden z reszty zablokowanych wątków. Na przykład jeśli wątek A blokuje rekord nr 1 i oczekuje na zablokowanie rekordu nr 2, a w tym samym czasie wątek B blokuje rekord nr 2 i oczekuje na zablokowanie rekordu nr 1, wówczas wątki te są zakleszczone.

zalogowany użytkownik interakcyjny

Użytkownik, który jest aktualnie zalogowany na komputerze, na którym działa program Transaction Server.

zasada działań

Kombinacja edycji sprawdzających ważność, weryfikacji logowań, wyszukiwań w bazie danych, założeń systemowych i przekształceń algorytmicznych składających się na sposób działania. Nazywana również *logiką działań.*

zdalny składnik

Składnik używany przez klienta na innym komputerze.

zmienna obiektowa

Zmienna zawierająca odwołanie do obiektu.

zwartość

Stan, w którym dane trwałe odpowiadają stanowi oczekiwanemu przez działania modyfikujące dane.