Getting Started with Microsoft Transaction Server

Microsoft® Transaction Server (MTS) is a component-based transaction processing system for developing, deploying, and managing high-performance, scalable, and robust enterprise, Internet, and intranet server applications. MTS defines a programming model for developing distributed, component-based applications. It also provides a run-time infrastructure and a graphical tool for deploying and managing these applications.

This section provides an overview of the new features in MTS, gives a brief tour of the documentation, and contains a glossary of terms. The following topics are contained in this section:

- What's New in MTS
- MTS Documentation Roadmap
- MTS Glossary
- MTS Utilities
- MTS Frequently Asked Questions (FAQs)

What's New in MTS

Microsoft Transaction Server (MTS) version 2.0 contains many new features that facilitate deploying robust and scalable Internet and intranet applications. This section provides a summaries of new features in MTS.

Complete integration with Internet Information Server (IIS) version 4.0

MTS 2.0 is tightly integrated with IIS 4.0, creating the best platform for business applications on the Web. New features from the integration of MTS and IIS include:

- Transactional Active Server Pages

  Scripts in Active Server Pages can now execute within an MTS-managed transaction. This extends the benefits of MTS transaction protection to the entire Web application.
- Crash Protection for IIS Applications

  IIS Web applications can now execute within their own MTS package, providing process isolation and crash protection for Web applications.
- Transactional Events

  Developers can embed commands into scripts on Active Server Pages, enabling customization of Web application response based on transaction outcome.
- Object Context for IIS Built-In Objects

  The MTS object context mechanism, which masks the complexity of tracking user state information from the application developer, can now track state information managed by the IIS built-in objects. This extends the simplicity of the MTS programming model to Web developers.
- Common Installation and Management

  MTS and IIS now share common installation and a common management console, lowering the complexity of deploying and managing business applications on the Web.

Support for the XA transaction protocol, including native support for Oracle

- MTS 2.0 supports the XA transaction protocol, which enables MTS applications to work with IBM DB2, Informix, and other XA-compliant databases running on Windows NT Server and non-Microsoft operating systems (including versions of UNIX) via ODBC.
- Vendors of XA-compliant databases are in the process of testing their ODBC drivers for XA interoperability with MTS 2.0; contact your database vendor for more information.
- MTS 2.0 includes a revision to Microsoft's ODBC driver for Oracle, enabling MTS applications to directly manage transactions with Oracle version 7.3 or greater via ODBC.

Desktop Operating System Support

MTS 2.0 can be used with both the Windows NT version 4.0 and Windows 95 operating systems. MTS support for Microsoft's desktop operating systems enables businesses to deploy stand-alone versions of their MTS applications.

- Windows 95 can be used as a development platform for developing MTS components prior to deployment on a Windows NT server. A Windows 95 computer can also be used as an administration client for a MTS server application running on Windows NT. In addition, MTS provides the runtime environment for the Personal Web Server (PWS), which runs on Windows 95.
- Due to the differences between Windows NT and Windows 95, MTS on Windows 95 does not support Microsoft Cluster Server or MTS role-based security. You cannot remotely administer a Windows 95 machine running MTS from a computer running Windows NT or Windows 95. Also, the Microsoft Visual Basic Scripting Edition (VBScript) samples demonstrating how to automate MTS administration are not installed with MTS on Windows 95 computers.

Microsoft Cluster Server Support

MTS 2.0 supports Microsoft Cluster Server (MSCS), which enables automatic failover of MTS

packages in a cluster. Automatic failover enables high availability for MTS applications.

Support for CICS and IMS transactions via LU 6.2 Sync Level 2

MTS 2.0 enables businesses to deploy MTS applications that support CICS and IMS transactions on MVS. MTS offers full support for beta 2 or later of Cedar, the SNA Server component that provides interoperability between MTS and CICS/MVS and IMS/MVS.

Administration Enhancements

MTS 2.0 provides administration enhancements that facilitate deploying and administering MTS packages, including:

- Snap-in MTS Explorer

  The MTS Explorer is now a Microsoft Management Console (MMC) snap-in. You can manage your MTS packages in the same administrative console as other products, such as IIS. For more information about MMC and using snap-ins, see the MMC documentation.
- Shut-down of individual server processes

  In MTS 2.0, you can shut down individual packages in the MTS Explorer without shutting down the MTS server process. Shutting down a package terminates that application's server process.
- Improved activation settings for packages

  In MTS 2.0, activation is only set at the package level. MTS no longer allows mixed activation for components in a package. In addition, you cannot run your package with the remote activation setting. Packages are either set at **Server** (local activation of components) or **Library** (client caller process activation of components).

  MTS 1.0 packages containing components that are all marked in-process will be automatically upgraded to a Library package when installing MTS 2.0. MTS 1.0 packages containing components that are all marked with the local activation setting will be automatically upgraded to a Server package. MTS 1.0 packages with mixed component activation must be manually configured as a Library or Server package.
- Multi-select capability for updating properties

  You can select and modify the properties of multiple items in the MTS Explorer at the same time.
- Move components between packages

  Components can be moved between packages by dragging and dropping the component from one package to another.

Programming enhancements

MTS 2.0 has made it easier to build MTS applications with programming enhancements that include:

- New and updated MTS sample applications

  Along with an updated Sample Bank application, MTS now offers two new sample applications. The Tic-Tac-Toe sample application demonstrates non-transactional components managing shared state in a simple multiuser game. The administrative sample scripts show you how to use the scriptable administration objects to automate procedures in the Explorer in a Windows Scripting Host script.
- Scriptable Administration Objects

  Package deployment and maintenance in the MTS Explorer can be automated using the scriptable administration objects. Using any Automation-compatible language, you can automate administrative procedures such as installing a package using a simple script.
- MTS application design and implementation documentation

  The *MTS Programmer's Guide* provides design and implementation guidance for creating an MTS application. Topics range from enacting business logic in components todiagnosing and debugging problems.

See Also

Road Map to the Administrator's Guide, MTS Overview and Concepts

MTS Documentation Roadmap

Microsoft Transaction Server (MTS) contains documentation that helps you learn how to design, build, deploy, and administer MTS applications.

Book	Description
Setting Up MTS	Describes how to set up MTS and MTS components, including instructions for accessing Oracle databases from MTS application and installing MTS sample applications.
Getting Started with MTS	Provides an overview of the new features in MTS, gives a brief tour of the documentation, and contains a glossary of terms.
Quick Tour of MTS	Provides an overview of MTS.
MTS Administrator's Guide	
Roadmap to the MTS Administrator's Guide	Describes the different ways to use the MTS Explorer to deploy and administer applications, and gives an overview of the MTS Explorer graphical interface.
Creating MTS Packages	Provides task-oriented documentation for creating and assembling MTS packages.
Distributing MTS Packages	Provides task-oriented documentation for distributing MTS packages.
Installing MTS Packages	Provides task-oriented documentation for installing and configuring MTS packages.
Maintaining MTS Packages	Provides task-oriented information for maintaining and monitoring MTS packages.
Managing MTS Transactions	Describes distributed transactions and the management of transactions using the MTS Explorer.
Automating MTS Administration	Provides a conceptual overview, procedures, and sample code explaining how to use the MTS scriptable objects to automate procedures in the MTS Explorer.
MTS Programmer's Guide	
Overview and Concepts	Provides an overview of the product and how the product components work together, explains how MTS addresses the needs of client/server developers and system administrators, and provides in-depth coverage of programming concepts for MTS components.
Building Applications for MTS	Provides task-oriented information for developing ActiveX™ components for

	MTS.
MTS Administrative Reference	Provides a reference for using the MTS scriptable objects to automate procedures in the MTS Explorer.
MTS Reference	Provides a reference for the MTS application programming interface (API).
MTS Administrative Reference	Provides a reference for using the MTS scriptable objects to automate procedures in the MTS Explorer.

MTS Utilities

MTS provides command-line utilities that you can use to automate certain tasks in a batch file (these utilities are available directly from the command prompt).

Command-Line Utilities

The following table is a quick reference to command-line utilities that are installed with MTS.

Utility	Function
MTXSTOP.exe	Shut down all MTS processes. This is the command-line version of the **Shut Down Server Processes** option on the **My Computer** right-click menu.
MTXTEST.exe	Test component marshaling code outside the MTS run-time environment.
MTXTSTOP.exe	Stop MTXTEST.exe. This tool is installed only with the development installation option.
SAMPDTCC.exe	Test the MS DTC installation with a sample client.
SAMPDTCS.exe	Test the MS DTC installation with a sample server.
MTXREREG.exe	Refresh all components registered on your computer. This is the command-line version of the **Refresh Components** option that you can access by right-clicking on a selected package.
MTXREPL.exe	Replicate an MTS server. Both the master and destination computers must be running.
TestOracleXAConfig .Exe	Test Oracle configuration to validate distributed transactions involving MTS components. If this utility fails, distributed transactions with Oracle databases will not work.

Windows NT Administrative Tools

Windows NT also provides several tools that you can use to administer your MTS applications. To use these tools, click the **Start** button, point to **Programs**, and then point to the **Administrative Tools (Common)** menu.

Tool		Function
	Event Viewer	In Windows NT, an event is any significant occurrence in the system or in a program that requires you to be notified. Event Viewer either notifies you or puts the event in a log. Refer to the Event Viewer as the first step in diagnosing a problem in your MTS application.
	Performance Monitor	Performance Monitor is a tool for monitoring the performance of your computer or other computers on a network.
	Server Manager	Server Manager displays a list of workstations and servers in your domain.
	User Manager for Domains	User Manager for Domains enables you to establish, delete, or disable domain user accounts. You can also set security policies and add user accounts to groups.

	Windows NT Diagnostics	Windows NT Diagnostics displays information about your computer's resources.

Caution Many of these Windows NT Administrative Tools require that you be logged onto that machine with administrative privileges.

See Also

Roadmap to the MTS Administrator's Guide

MTS Frequently Asked Questions (FAQs)

Frequently Asked Questions for the Microsoft Transaction Server are located on
http://www.microsoft.com/support/transaction/content/faq/.

Quick Tour of Microsoft Transaction Server

Microsoft Transaction Server (MTS) is a component-based transaction processing system for developing, deploying, and managing high-performance, scalable, and robust enterprise, Internet, and intranet server applications.

The following sections introduce the features of Microsoft Transaction Server:

- What is Microsoft Transaction Server?
- Microsoft Transaction Server Run-Time Environment
- Microsoft Transaction Server Explorer
- Microsoft Transaction Server APIs
- Microsoft Transaction Server Sample Applications

What Is Microsoft Transaction Server?

MTS is a component-based transaction processing system for building, deploying, and administering robust Internet and intranet server applications. In addition, MTS allows you to deploy and administer your MTS server applications with a rich graphical tool (MTS Explorer).

MTS provides the following features:‹

- The MTS run-time environment.
- The MTS Explorer, a graphical user interface for deploying and managing application components.
- Application programming interfaces and *resource dispensers* for making applications scalable and robust. Resource dispensers are services that manage non-durable shared state on behalf of the application components within a process.
- Three sample applications that demonstrate how to use the application programming interface (API) to build MTS components, and use scriptable administration objects to automate deployment procedures in the MTS Explorer.

The MTS programming model provides a framework for developing components that encapsulate business logic. The MTS run-time environment is a middle-tier platform for running these components. You can use the MTS Explorer to register and manage components executing in the MTS run-time environment.

The three-tiered programming model provides an opportunity for developers and administrators to move beyond the constraints of two-tier client/server applications. You have more flexibility for deploying and managing three-tiered applications because:

- The three-tier model emphasizes a logical architecture for applications, rather than a physical one. Any service may invoke any other service and may reside anywhere.
- These applications are distributed, which means you can run the right components in the right places, benefiting users and optimizing use of network and computer resources.

See Also

MTS Documentation Roadmap

Microsoft Transaction Server Run-Time Environment

The MTS run-time infrastructure makes application development, deployment, and management easy by providing the application developer and system administrator a comprehensive but easy-to-use set of system services that include:

- Distributed transactions. A *transaction* is a unit of work that is done as an atomic operation – that is, the operation succeeds or fails as a whole.
- Automatic management of processes and threads.
- Object instance management.
- A distributed security service to control object creation and use.
- A graphical interface for system administration and component management.

Application developers who rely upon these system services to make their applications scalable and robust can focus on solving their business problems rather than on developing a system infrastructure.

MTS works with any application development tool capable of producing ActiveX dynamic-link libraries (DLLs). For example, developers can use Microsoft Visual Basic, Microsoft Visual C++, Microsoft Visual J++, or any other ActiveX tool to develop MTS applications.

MTS is designed to work with a wide variety of resource managers, including relational database systems, file systems, and document storage systems. This allows developers and independent software vendors to select from a wide range of resource managers and to easily use two or more resource managers within a single application while enjoying the benefits of local or distributed transactions.

See Also

MTS Overview and Concepts

Microsoft Transaction Server Explorer

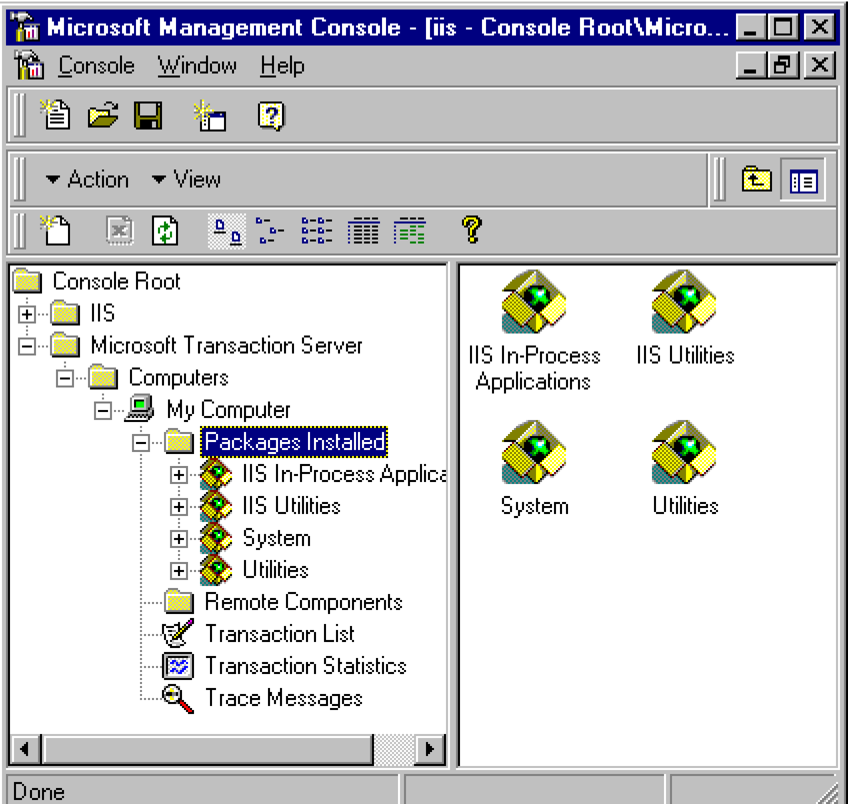
The MTS Explorer is a graphical user interface for managing and deploying MTS components. System and web administrators as well as developers can use the MTS Explorer to administer, distribute, install, deploy, and test packages. Developers use the MTS Explorer to assemble components into pre-built packages, distribute and test components in the MTS environment. Administrators or developers also use the Explorer to install, deploy, and maintain components and packages. In addition, the Explorer allows you to monitor and manage transactions for your transactional components.

The Explorer hierarchy depicts how the following items in the run-time environment are organized:

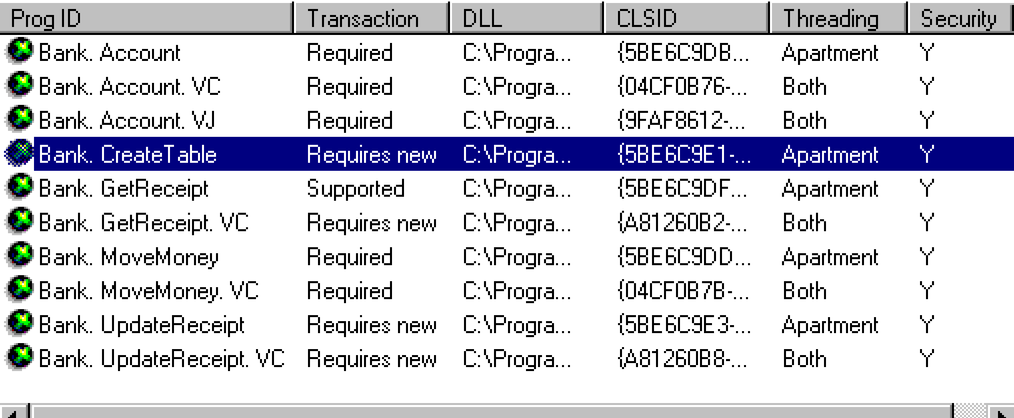
- Computers
- Packages
- Components
- Roles
- Interfaces
- Methods

MTS packages are installed on computers, contain components, and define roles. Components in a package define interfaces and methods. You can use special purpose windows to view transaction and trace message information.

The following diagram shows how the hierarchy is displayed in the left pane of the Explorer:

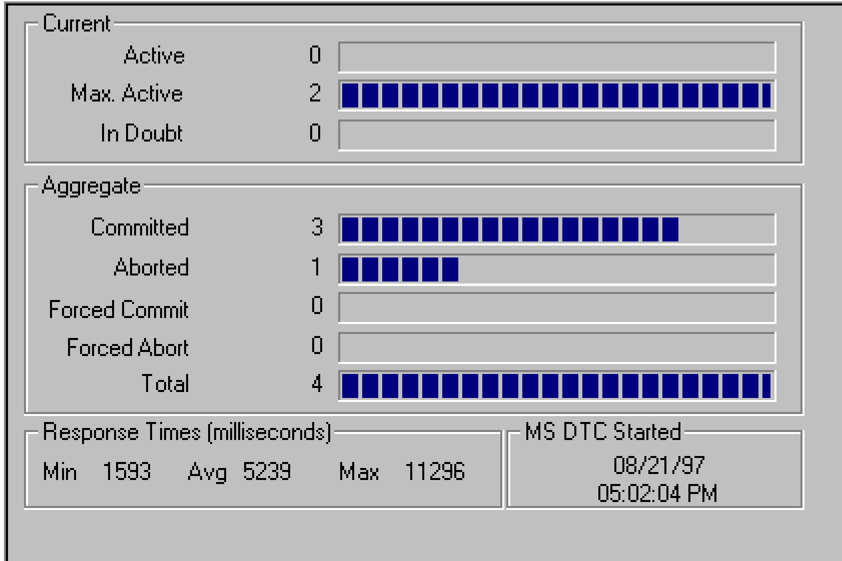


You can use the properties window to view the properties of components in a package(see the following diagram) or to view packages installed on a computer.



Prog ID	Transaction	DLL	CLSID	Threading	Security
Bank. Account	Required	C:\Progra...	{5BE6C9DB...	Apartment	Y
Bank. Account. VC	Required	C:\Progra...	{04CF0B76-...	Both	Y
Bank. Account. VJ	Required	C:\Progra...	{9FAF8612-...	Both	Y
Bank. CreateTable	Requires new	C:\Progra...	{5BE6C9E1-...	Apartment	Y
Bank. GetReceipt	Supported	C:\Progra...	{5BE6C9DF...	Apartment	Y
Bank. GetReceipt. VC	Requires new	C:\Progra...	{A81260B2-...	Both	Y
Bank. MoveMoney	Required	C:\Progra...	{5BE6C9DD...	Apartment	Y
Bank. MoveMoney. VC	Required	C:\Progra...	{04CF0B7B-...	Both	Y
Bank. UpdateReceipt	Requires new	C:\Progra...	{5BE6C9E3-...	Apartment	Y
Bank. UpdateReceipt. VC	Requires new	C:\Progra...	{A81260B8-...	Both	Y

You can also use the **Transaction Statistics** window to view summary descriptive statistics for recent transactions.



Navigating the MTS Explorer

The MTS Explorer window has a left pane that displays a hierarchy and a right pane that displays the contents of the item you click in the left pane. The hierarchy is a tree structure that contains folders and all the items you can configure with the MTS Explorer.

You can navigate through the MTS Explorer hierarchy by double-clicking a folder or item in the right pane to expose its contents. You can also view those contents by clicking the folder or item in the left pane, which displays the contents in the right pane. To expand any item in the hierarchy, click the plus (+) sign beside it, and the MTS Explorer displays the hierarchy of that item in the left pane. Double-clicking a folder or item in the left pane will also display its contents in the right pane so that you can switch between the expanded and collapsed views of the hierarchy.

Use arrows keys in either pane to select an item. Pressing the ENTER key will display the contents of an item. You can move between the Explorer's two panes by pressing the TAB key.

Note that on Windows 95, the tree in the left pane in the MTS Explorer does not appear. To navigate, double-click icons to move down the hierarchy and then click the **Up one level** toolbar button to move up the hierarchy. For more information about using the MTS Explorer on Windows 95, see the Roadmap to the MTS Administrator's Guide.

Setting or Viewing an Item's Properties

Basic information about an item that has been added to the MTS Explorer hierarchy is displayed in the item's property sheet. What information appears on a property sheet varies from item to item. For example, the property sheet for a computer item contains the computer's name, location of the log file, and update settings, whereas the property sheet for a package item contains information regarding security and other process-specific settings.

You can view property sheets by selecting an item and choosing the **Properties** command from the **Action** menu, or right-clicking the item and selecting **Properties**. Each property sheet is divided into individual pages, which you can access by clicking the appropriate tab.

Monitoring and Resolving Transactions

Extensive support for transaction processing is a major feature of the MTS run-time environment. Microsoft Distributed Transaction Coordinator (MS DTC) provides the support for transaction processing in MTS. MS DTC is a Windows NT service that MTS uses to ensure that all parties in a transaction are in agreement before it is finalized.

Transaction support is provided in the MTS Explorer hierarchy through three windows:

You use the Transaction List window to monitor the state of an active transaction.

You use the Transaction Statistics window to view summary statistics for recent transactions.

You use the Trace Messages window to view trace messages relating to transaction processing.

See Also

Roadmap to the MTS Administrator's Guide

Microsoft Transaction Server APIs

You can use MTS application programming interfaces (APIs) to develop scalable and robust applications that take advantage of the features of the MTS run-time environment, and to automate administration of packages and components.

Developing Client Applications

Client applications that run outside the MTS run-time environment instantiate MTS objects by using the standard COM library functions (**CoCreateInstance** in C++; the Visual Basic **CreateObject** method performs the same function).

Developing Components

If you are developing MTS components (server components that will be registered in the MTS run-time environment), you can use the MTS **IObjectContext**, **ISharedPropertyGroupManager**, **ISharedPropertyGroup**, and **ISharedProperty** interfaces to:

- Declare that an object's work is complete
- Prevent a transaction from being committed
- Create other MTS objects
- Include other objects' work within the scope of the current object's transaction
- Determine if a caller is in a particular role
- Determine if security is enabled

Automating MTS Administration

You can automate administration of packages and components using the MTS administrative objects. Using Visual Basic, Visual Basic Scripting Edition (VBScript), or any other Automation-compatible language, you can automate procedures in the MTS Explorer ranging from installing a prebuilt package to enumerating through related collections.

See Also

Roadmap to the MTS Administrator's Guide, MTS Overview and Concepts,MTS Reference, MTS Administrative Reference

Microsoft Transaction Server Sample Applications

In addition to the documentation, MTS includes useful sample applications that are valuable learning tools. You can copy any part of them into your own applications and modify them as necessary.

Throughout the MTS Programmer's Guide, sample code and applications illustrate MTS programming techniques. Many of the files for these applications are included with your installation. You can find the source files for the applications in the \Samples folder of your MTS installation.

MTS provides the following sample applications.

Sample	Description
Sample Bank	Sample Bank is a simple transactional database application that demonstrates how to use the MTS application programming interfaces
Tic-Tac-Toe	Tic-Tac-Toe is a simple multiuser game that shows nontransactional components managing shared state.
Administrative Sample Scripts	The administrative object scripts demonstrate how to automate MTS Explorer procedures using VBScript.

See Also

Installing MTS Development Samples and Documentation, Setting Up the MTS Sample Bank Application, Setting Up the MTS Tic-Tac-Toe Sample Application, Setting Up the MTS Administrative Sample Scripts, MTS Overview and Concepts,Visual Basic Script Sample for Automating MTS Administration

Setting Up Microsoft Transaction Server

Welcome to Microsoft Transaction Server (MTS), a powerful environment that makes it easier to develop and deploy high performance, scalable, and robust enterprise, Internet, and intranet applications. MTS defines an application programming model for developing distributed, component-based applications. It also provides a run-time infrastructure for deploying and managing these applications.

Refer to the following topics to learn how to install and set up MTS:

- MTS System Requirements
- Installing MTS Development Samples and Documentation
- Configuring Your MTS Server
- Configuring MTS with Microsoft Cluster Server
- Setting Up MTS to Access Oracle
- Setting Up the MTS Sample Bank Application
- Setting Up the MTS Tic-Tac-Toe Sample Application
- Setting Up the MTS Administrative Object Sample Scripts
- Getting Assistance While You Work with MTS

MTS System Requirements

This section describes the hardware and software requirements for installing MTS, discusses other setup considerations, and describes MTS support on Alpha platforms.

You can install MTS on your computer by:

- Using the Windows NT 4.0 Option Pack to install MTS with Internet Information Server (IIS) or other Option Pack components.
- Using the Windows NT 4.0 Option Pack to install MTS without any other Option Pack components.

MTS runs on Windows NT and Windows 95 with DCOM support.

To install MTS without other Option Pack components:

1. Run the Option Pack Setup program. Choose **Custom** install.
2. Uncheck all Option Pack components.
3. Select, but do not click the check box, for Transaction Server.
4. Click **Show Subcomponents**.
5. Check **Transaction Server Core Components**. This will also check Microsoft Management Console.

   Note that choosing the Development Option also installs the Data Access components.
6. Click **OK** and continue the setup program.

Hardware Requirements

Before you install Microsoft Transaction Server, make sure that your computer meets the following minimum requirements:

- Any Windows NT or Windows 95 with DCOM support i386-compatible computer or Alpha AXP™ computer. See the MTS Release Notes for the latest information on hardware requirements.
- A hard disk with a minimum of 30 megabytes available space for a full installation.
- A CD drive.
- Any display supported by Windows NT version 4.0 or Windows 95.
- At least 32 megabytes of memory.
- A mouse or other suitable pointing device.

You can run MTS setup unattended using the Windows NT 4.0 Option Pack setup. Before you run setup unattended, modify the setup file with your setup selections. The unattended setup file (unattend.txt) is located in your \MTS directory.

To run setup unattended, click the **Start** button, select **Run** and type the following:

```
setup /u:unattend.txt.
```

Software Requirements

MTS software requirements include:

- Before you install MTS, you must install Microsoft Windows NT version 4.0 or later, or Windows 95 with DCOM support on your computer. Failure to install DCOM support for Windows 95 before setting up MTS will result in the following error message:

  ```
  Setup library mtssetup.dll could not be loaded or the function
  MTSSetupProc could not be found.
  ```

  You can install DCOM support for Windows 95 from http://www.microsoft.com/oledev. Note that DCOM support for Windows 95 is installed by Internet Explorer 4.0.

If you uninstall MTS on a Windows 95 computer, do not uninstall DCOM support for Windows 95. The Microsoft Distributed Transaction Coordinator (MS DTC), which is installed but not removed by the MTS setup program, requires DCOM support on Windows 95.

- If you want to remotely administer a Windows NT computer from a Windows 95 computer, you must install the Remote Registry service for Windows 95. The Remote Registry service allows you to change registry entries for a remote Windows NT computer (given the appropriate permissions). To obtain the Remote Registry service, go to the \Admin\Nettols\RemoteReg subdirectory on the Windows 95 CD. Read the Regserv.txt file for instructions on installing the Remote Registry service.
- If you want to use a Microsoft Windows 95 client with MTS, install DCOM for Windows 95. For the latest information on DCOM support for Windows 95, see http://www.microsoft.com/oledev/.
- If you are using Windows NT Server, you must install Windows NT Service Pack 3. You can download Windows NT Service Pack 3 from http://www.microsoft.com/support.
- If you want your components to access databases, use Microsoft SQL Server, version 6.5 or later, or another Microsoft Transaction Server-compatible database. Note that SQL Server requires the Windows NT operating system.
- It is strongly recommended that you install the SQL Server Service Pack 3 on computers running MTS software. This Service Pack resolves several known problems. You can install SQL Server Service Pack 3 from http://www.microsoft.com/support/.
- If you plan to create components with Microsoft Visual Basic, use Microsoft Visual Basic, Enterprise Edition, version 4.0 or later. It is strongly recommended that you use Visual Basic version 5.0 to build your MTS components.
- If you plan to create components with Microsoft Visual C++®, use version 4.1 or later with the Active Template Libraries (ATL), version 1.1 or later. If you use Visual C++ version 4.1, you must have the Win32 SDK.
- If you plan to create and run Java components with MTS, use the Microsoft Virtual Machine for Java installed with IE 4.0 or later. You can download Internet Explorer from http://www.microsoft.com/ie/.
- If you plan to create Internet applications, you must use Microsoft Internet Information Server version 4.0 or later and Microsoft Internet Explorer version 4.0 or later.
- To use MTS with an Oracle 7.3.3 or an Oracle 8 database, install Oracle patch 7.3.3.2.0 or later from Oracle. To obtain this patch your must contact the Oracle support organization. In addition, please see the Setting Up MTS to Access Oracle topic, which provides essential information for making Oracle work with Microsoft Transaction Server.

Use the Advanced Data Connector 1.1 Client

Windows NT 4.0 Option Pack installs Advanced Data Connector (ADC) version 1.5. The ADC 1.5 client only works with the Internet Explorer (IE) 4.0. The ADC 1.5 client will not work with any version of Internet Explorer before IE 4.0.

For users of the MTS Adventure Works sample application, check http://www.microsoft.com/adc for an updated ADC 1.1 client that will work with Internet Explorer version 3.0 or later. You will need to recompile the MTS Adventure Works sample to set a reference to the new ADC 1.5 recordset library (Msador15.dll).

Installing SQL Server with MTS

If you install both SQL Server and MTS, you must install SQL Server first. If you later reinstall SQL Server, you must then reinstall MTS. This limitation will be eliminated in a future release of SQL Server.

Installing MTS 1.0 over MTS 2.0

To install MTS 1.0 on a computer with MTS 2.0 installed, you must remove the following files from

your system directory before running MTS 1.0 setup:

%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

Uninstall Option Removes User-Defined Packages

If you are using a previous version of MTS and would like to retain your user-defined packages, do not use the **Uninstall** option in the setup program or the **Add/Remove Programs** service icon in the Control Panel. Uninstalling MTS results in the removal of all user-defined packages. To preserve user-defined packages when upgrading, install MTS over your existing version of MTS.

Incorrect System Package Identity Passwords

If you incorrectly specify a password for the System Package Identity on setup, the Microsoft Transaction Server Explorer will not run. You will get a 80008005 error (server exec failure) back from the catalog server (System package). To fix a bad system password, reinstall Microsoft Transaction Server.

In addition, if you incorrectly specify the password on a System Package Identity, the package won’t run. You will get the same 80008005 error. But this time, the client will see this failure HRESULT on a **CoCreateInstance** or equivalent. To fix this, go back to the Explorer and adjust the package's identity (through the package's **Identity** tab). User IDs are verified, but passwords are not because of security restrictions in Windows NT.

Alpha Platforms

MTS is supported on Alpha AXP™ computers. However, this release of Microsoft Transaction Server for Alpha platforms does not include the following:

- Microsoft Visual J++ components and samples
- Support for accessing Oracle databases

You can find sample Microsoft Visual C++ and Visual Basic components as well as the administrative sample scripts in the \Samples folder.

For the latest information on Alpha platform support, see http://www.microsoft.com/transaction.

To validate your MTS installation on Alpha platforms, you can run the Sample Bank application using two computers, an Alpha computer (as a server), and an X86 computer (as a client). See the Setting Up the MTS Sample Bank Application topic for more details.

Installing MTS Development Samples and Documentation

You install Microsoft Transaction Server (MTS) development samples and documentation by using the **Custom** setup option. The following list describes the MTS components installed with each setup option.

- **Minimal**

  Installs the MTS run-time environment and MTS Explorer.
- **Typical**

  Installs the MTS run-time environment, MTS Explorer, and MTS core documentation.
- **Custom**

  Installs the MTS run-time environment, MTS Explorer, MTS core documentation, MTS development samples, and MTS development documentation.

It is highly recommended that you use the **Custom** install option in order to obtain MTS development samples and documentation. Reviewing the MTS development samples and documentation will provide a more thorough understanding of the packages and components that you manage. For example, you can use the MTS Sample Bank and Tic-Tac-Toe development samples to practice deploying and administrating packages in the MTS Explorer. The sample applications also help you confirm that you have correctly installed MTS.

The MTS development documentation explains the design decisions that determine how components are packaged, such as activation settings and transactional properties. Along with discussions of MTS programming concepts (such as sharing state), the *MTS Programmer's Guide* also contains an tutorial that describes how to install and configure the Sample Bank application.

To obtain MTS development samples and documentation after you have already installed the Minimum or Typical install:

1. Open the **Start** menu, click **Settings**, and then **Control Panel**.
2. Select the **Add/Remove Programs** option, click **Microsoft Transaction Server**, and then the **Add/Remove** button.

   Note that if you uninstall MTS, user-defined packages will be removed. If you want to maintain your user-defined packages, install MTS without removing the previous installation.
3. From the setup program, select the **Reinstall/Add** button.
4. Select the **Microsoft Transaction Server** option, and click the **Show Subcomponents** button.
5. Verify that all MTS subcomponent checkboxes have been selected.
6. Click **OK**.

Configuring Your MTS Server

After you install MTS, configure your MTS server so that you can deploy and manage MTS packages using the MTS Explorer. Before you start deploying and administering packages, set your MTS server up for deployment by doing the following:

- Configuring roles and package identity on the System package
- Setting up computers to administer

Configuring Roles on the System Package

You must map the System package Administrator role to the appropriate user in order to safely deploy and manage MTS packages. When MTS is installed, the System package does not have any users mapped to the administrator role. Therefore, security on the System package is disabled, and any user can use the MTS Explorer to modify package configuration on that computer. If you map users to System package roles, MTS will check roles when a user attempts to modify packages in the MTS Explorer.

By default, the System Package has an Administrator role and a Reader role. Users mapped to the Administrator role of the System package can use any MTS Explorer function. Users that are mapped to the Reader role can view all objects in the MTS Explorer hierarchy but cannot install, create, change, or delete any objects, shut down server processes, or export packages. For example, if you map your Windows NT domain user name to the System Package Administrator role, you will be able to add, modify, or delete any package in the MTS Explorer. If MTS is installed on a server whose role is a primary or backup domain controller, a user must be a domain administrator in order to manage packages in the MTS Explorer.

For more information on how to map users to roles, see the Mapping MTS Roles to Users and Groups topic.

You can also set up new roles for the System package. For example, you can configure a Developer role that allows users to install and run packages, but not delete or export them. The Windows NT user accounts or groups that you map to that role will be able to test installation of packages on that computer without having full administrative privileges over the computer. For more information on setting up new roles, see the Adding a New MTS Role topic.

Once you have configured roles for your computer's System package, enable authorization checking by selecting the check box in the Package Security property sheet. See the Enabling MTS Package Security for a complete description of how to enable authorization checking.

Note that the following MS DTC administrative functions do not use the System package or the System package role:

- Transaction Statistics window
- Transaction List window
- Trace Messages window
- Start/Stop MS DTC commands

Setting Up Computers to Administer with the MTS Explorer

By default, the computer on which you install MTS is managed in the MTS Explorer as "My Computer". You can also use the MTS Explorer to manage other computers. Add any new computers that you need to administer to the Computers folder in the Explorer by selecting the Computer icon and doing one of the following:

- Selecting **New** from the **Action** menu
- Clicking the **Create a new object** icon on the MTS Explorer toolbar
- Right-clicking My Computer and choosing **New** and then **Computer**

Then enter a computer name in your Windows NT domain in the dialog box to add the remote computer as a top-level folder. You must be mapped to the Administrator role on the remote computer.

For more information on managing objects in the MTS Explorer Hierarchy, see the MTS Explorer Hierarchy topic.

Important You can not remotely administer MTS on a Windows 95 computer from MTS on a Windows NT server.

Configuring MTS with Microsoft Cluster Server

Microsoft Cluster Server (MSCS) is the clustering solution for computers running Windows NT Server. MSCS version 1.0 supports clusters of two specially linked servers running Windows NT Server. If one server in a cluster fails or is taken offline, the other server takes over the failed server's operations.

If you wish to use Microsoft Transaction Server with MSCS, first install Windows NT Server 4.0 Enterprise. Then install Microsoft Cluster Server 1.0. Finally, install Microsoft Transaction Server 2.0 as described below.

Note MTS 2.0 cluster support does not work with pre-release versions of Microsoft Cluster Server 1.0. Installing MTS 2.0 with a pre-release version of Microsoft Cluster Server will prevent MSCS from functioning properly.

For more information on MSCS, see Windows NT Books Online.

To install Microsoft Transaction Server 2.0 with Microsoft Cluster Server 1.0

1. Install Windows NT Server 4.0 Enterprise.
2. Install MSCS 1.0 on all computers in the cluster.
3. Use the MSCS Cluster Administrator to configure a Group to contain a Network Name Resource and a Shared Disk Resource.
4. Install Microsoft Transaction Server 2.0 on the node that owns the Group configured above. See Setting Up Microsoft Transaction Server for more information.
5. When MTS setup detects that MSCS is present on the system, it will display a dialog box asking you to specify the name of the Virtual Server on which Microsoft Distributed Transaction Coordinator should be installed. Specify the name of the Network Name Resource configured in step 3.
6. In the same dialog box, specify the location for the MS DTC log file on the shared disk configured in step 3.
7. Click **NEXT** to continue MTS setup.
8. Install MTS on the second computer in the cluster. You will not be prompted for the virtual server and log file location during setup.

Do not run MTS setup in parallel on cluster nodes. Completely install MTS on one node, then install MTS on the second node without rebooting the first node. When both nodes have MTS installed, reboot them.

Note You do not not need to uninstall Microsoft Transaction Server 1.0 before Microsoft Transaction Server 2.0 with Microsoft Cluster Server 1.0. Follow the instructions for installing MTS 2.0 with MCS 1.0.

To upgrade from Microsoft Transaction Server 1.1 to Microsoft Transaction Server 2.0 with Microsoft Cluster Server

1. Delete all Microsoft Transaction Server package resources that you created when using Microsoft Transaction Server 1.1 with Microsoft Cluster Server.
2. Run the Microsoft Transaction Server 2.0 setup program on the cluster node that owns the Microsoft Distributed Transaction Coordinator resource. This will automatically perform the upgrade on this node. Ensure that Microsoft Transaction Server setup runs to completion before starting the next step.
3. Run the Microsoft Transaction Server 2.0 setup program on the second node of the cluster.
4. Reboot both cluster nodes.

Uninstalling Microsoft Cluster Server on an MTS Server

If you want to uninstall Microsoft Cluster Server from one of the nodes in a cluster, follow these

steps:

1 Using the Cluster Administrator to take the Microsoft Distributed Transaction Coordinator resource offline.
2 Uninstall Microsoft Cluster Server from the node and reboot the node.
3 Change the location of the Microsoft Distributed Transaction Coordinator log file using the Transaction Server Explorer. The log file must be placed on a non-shared disk. Failing to do this may result in a corrupted log file or access violations in Microsoft Distributed Transaction Coordinator.

Resetting the MS DTC Log File on Clustered Servers

To reset the MS DTC log file on a clustered server, you must run the MTS Explorer on the node that currently owns the shared disk containing the log file.

Starting and Stopping MS DTC on Clustered Servers

On a clustered server, you can start and stop MS DTC by either using the MTS Explorer or using the MSCS administrative utility. Using the "net start msdtc" or "net stop msdtc" command does not work. To start and stop MS DTC on an MSCS cluster from the command line, use "msdtc -start" and "msdtc -stop", respectively.

Calling MTS Objects after MSCS Failover

Microsoft Transaction Server client applications must always release all references to MTS components on the failing node, and re-instantiate the components. The components are instantiated on the second node.

Using the Remote Components Folder with Clustered Servers

You can use the Remote Components folder to pull components from a Microsoft Cluster Server (MSCS) node by following these steps:

1 On the server computer from which you will pull components, right-click on My Computer and select **Properties**.
2 Click the **Options** tab and enter the name of the virtual server in the **Remote Server Name** field. Click **OK**.
3 On the client computer to which you will pull components, add the MSCS computer to the Computers folder using the physical name of the server.
4 Select the Remote Components folder and choose **New** from the **Action** menu. You can also right-click the Remote Components folder, select **New** and then **Component**.
5 Follow the instructions in the remote component installation wizard. Note that during the installation of remote components, the physical name of the server will be displayed rather than the virtual name (as specified in the **Remote Server Name** field of the server computer).
6 Select the Remote Components folder and click the Property View button on the MTS Explorer toolbar. The Server column in the Property View window displays the virtual server name.

Note If you change the remote server name on the MSCS computer, you must also reinstall the remote components on client computers.

Replicating Empty Packages

MTS replication does not replicate empty packages.

Setting Up MTS to Access Oracle

You can enable transactional MTS components to access an Oracle 7.3.3 database through ODBC. MTS works with Oracle 7 Workgroup Server for Windows NT, Oracle 7 Enterprise Server for Windows NT, Oracle 7 Enterprise Servers on UNIX, and Oracle Parallel Server on UNIX.

Your MTS component may access an Oracle 8 database on either Windows NT or Unix provided your Microsoft Transaction Server component uses the Oracle 7 client software. MTS does not support Oracle 8 client software.

This section includes the following topics:

Required Software

Setting Up Oracle Support

Testing Installation and Configuration of MTS Support for Oracle

Validating Oracle Installation and Configuration Using the Sample Bank Application

Known Limitations of MTS Support for Oracle

Required Software

Refer to the following table for a list of the software required to access an Oracle database from MTS components running on either the Windows NT or UNIX platform.

Component	Version
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

Important Earlier versions of the software will not work properly. Please ensure you install the correct versions of the software. Failing to do this is by far the most common source of problems when trying to use MTS with Oracle.

Oracle for Windows NT

You must install either the Oracle 7.3.3 Workgroup Server release for Windows NT or the Oracle 7.3.3 Enterprise Server release for Windows NT. The Oracle 7.3.2 and earlier releases of Oracle for Windows NT are not supported and will not work in conjunction with MTS transactions.

You must install Oracle 7.3.3 patch release 2 or later. This patch is required for all Oracle 7.3.3 clients accessing an Oracle 7.3.3 or Oracle 8 database. Oracle patch release 2 contains fixes that are required to make Oracle XA transaction support work properly on Windows NT. The Oracle 7.3.3 release will not work with MTS unless Oracle 7.3.3 patch release 2 or later is installed.

Note If you encounter problems setting up Oracle patch release 2 on Windows 95, contact Oracle for support.

To obtain Oracle 7.3.3 patch releases from the Oracle customer support organization you must submit a problem report to the Oracle Customer Support Organization. These patch releases were not available from the Oracle public web site at the time this note was written.

Oracle SQL*Net

You must install the Oracle SQL*Net 2.3.3 release for Windows NT. You can obtain this release from Oracle. Earlier versions of Oracle SQL*Net may not work.

Oracle OCIW32.DLL

You must ensure that the correct version of the Oracle OCIW32.DLL is installed. Be very careful to check the version installed on your computer.

The correct version of the Oracle OCIW32.DLL is:

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

The improper version of the Oracle OCIW32.DLL is:

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

You can obtain the correct version of this DLL from the Oracle 7.3.3 installation CD from the \WIN32\V7\RSF73 directory.

Oracle for UNIX

In order for transactional MTS components to access an Oracle database on UNIX, you must install the Oracle 7.3.3 release (or later) for that UNIX platform. In most cases, you will also be required to install an Oracle 7.3.3 patch release for Oracle on UNIX.

You must check with Oracle Customer Support to determine if an Oracle 7.3.3 patch release is required for your UNIX platform. Explain that you are going to access your Oracle database on UNIX using the new XA transaction support that is now included in the Oracle 7.3.3 release on Windows NT.

The following patch releases are known to work:

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

Microsoft Transaction Server 2.0

You must install Microsoft Transaction Server 2.0 if you wish to access an Oracle database using MTS.

Microsoft ODBC Driver for Oracle

The Microsoft ODBC 2.0 Driver for Oracle (MSORCL32.DLL) is required. The Windows NT 4.0 Option Pack program automatically installs this DLL.

If you wish to access an Oracle database, we strongly suggest that you use the new Microsoft ODBC Driver for Oracle 2.0 even if you do not require transaction support. This new driver offers better performance than the ODBC 1.0 driver it replaces. The ODBC 1.0 driver serialized all activity at the driver level; requests were single-threaded through the driver. The ODBC 2.0 driver serializes all activities at the connection level. This allows different database connections to be used in parallel.

ActiveX Data Objects (ADO)

If your applications use ADO, you must install the ADO version 1.5. Earlier ADO releases will not work with the new ODBC 3.5 Driver Manager. ADO 1.5 is included in the Windows NT 4.0 Option Pack setup program.

Setting Up Oracle Support

To set up Oracle support for MTS transactional components

1. Install the Oracle 7.3.3 release on Windows NT.

   If your Oracle database is located on a UNIX system, install the Oracle 7.3.3 release on that system.

2. Install the Oracle 7.3.3 patch 2 or later on Windows NT. The resulting Oracle version will be Oracle 7.3.3.2 or later depending upon which Oracle patch you install. You must install Oracle 7.3.3 patch 2 or later if you wish to access any Oracle 7 or Oracle 8 database on either Windows NT or Unix. These Oracle patches correct problems that affect Oracle clients.

   If you are using UNIX, install any Oracle 7.3.3 patch releases that are required for your UNIX system. I

3. Ensure that the correct version of the Oracle OCIW32.DLL is installed as described in the Required Software section.

4. Install the Microsoft Transaction Server 2.0 version 3.0, which automatically installs the following:
   - Microsoft Transaction Server 2.0, including the Microsoft OCI Interface
   - Microsoft ODBC 3.5
   - Microsoft ODBC 2.0 driver for Oracle
   - ADO 1.5

1 Delete the DTCXATM.LOG. Use the Explorer to locate and delete this file if found. Note that the Microsoft Distributed Transaction Coordinator service must be stopped before the DTCXATM.LOG file can be deleted.

5 Enable Oracle XA Support

To enable an Oracle database to work with MTS transactions

1. The system administrator must create views known as V$XATRANS$. To do this, the administrator must run an Oracle-supplied script, named "xaview.sql". This file can usually be found in C:\ORANT\RDBMS73\ADMIN.
2. The system administrator must grant SELECT access to the public on these views.

   ```
   Grant Select on V$XATRANS$ to public.
   ```

3. In the Oracle Instance Manager, click **Advanced Mode** on the **View** menu and select **Initialization Parameters** in the left pane.
4. In the right pane, select **Advanced Tuning** and increase the "distributed_transactions" parameter to allow for more concurrent MTS transactions to update the database at a single time.

See your Oracle Server documentation for more information about configuring Oracle support for XA transactions.

Testing Installation and Configuration of MTS Support for Oracle

After installing and configuring Oracle support, you should validate your Oracle installation using the Oracle test program installed with MTS. The Oracle test program uses Oracle's OCI XA interfaces in much the same way that MTS uses them.

The Oracle test program determines whether you can connect to an Oracle database using Oracle's XA facility. The Oracle test program uses standard Oracle interfaces and transaction facilities. It makes no use of Microsoft Transaction Server or Microsoft Distributed Transaction Coordinator. Therefore, failure of the test program indicates that your Oracle is installed or configured improperly.

Reinstall and reconfigure Oracle, or contact your Oracle representative.

To run the Oracle test program

1. Verify that you have installed all of the correct versions of the software as described in Required Software.
2. Create an ODBC DSN that refers to your Oracle database. Ensure that your DSN uses the new Microsoft Oracle ODBC 2.0 driver.
3. Ensure that you have enabled Oracle XA support.
4. Delete all existing Oracle trace files from the machine containing the MTS components that access the Oracle database. The easiest way to do this is to use the Windows Explorer to locate and delete all *.TRC files.
5. Delete the DTCXATM.LOG file, if found, from the computer hosting the MTS components that access the Oracle database. Use the Windows Explorer to locate and delete the DTCXATM.LOG file (if it is located on your computer).
6. From the MS-DOS Command Prompt run the Oracle test program (TestOracleXaConfig.exe) and supply your Oracle server user ID, password, and server name. For example:

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

   If you run the test program with no parameters, the program will display help information that describes the required parameters. The test program will display information about each Oracle operation performed and will indicate whether each operation was successful.
7. If the Oracle test program is able to connect to your Oracle database server without error, then it is very likely that MTS will work with Oracle also. If the Oracle test program reports any errors, follow these steps:
   - Document the exact error message that the Oracle test program displays.
   - Examine the Oracle trace file produced when running the Oracle test program. The Oracle trace information is located in the *.TRC file. The Oracle trace file contains extended error information that is extremely helpful in diagnosing problems.
   - Contact your Oracle support representative for assistance.

Validating Oracle Installation and Configuration Using the Sample Bank Application

After you have validated your Oracle installation and configuration using the Oracle test program, you can use the Sample Bank Application supplied with Microsoft Transaction Server to ensure that Microsoft Transaction Server can access your Oracle database.

To validate Oracle support using Sample Bank

1. Ensure that you have verified that your Oracle support is installed and configured correctly using the Oracle test program provided by MTS.
2. On the server, create a table named "Account". The following example demonstrates how to set up the Account table.

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

3. Populate the table with at least two rows. The following table illustrates how to populate the table.

AccountNo	Balance
1	1000
2	1000

4. Create a file DSN using the ODBC configuration utility. Name the file DSN "MTSSamples". Then manually update the DSN file to add the password of the user. The following example

demonstrates how to add the user password to a file DSN.

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

5 Save the file DSN and run the Sample Bank client.

Known Limitations of MTS Support for Oracle

ADO 1.5 Beta Release Is Required When Using ADO with ODBC 3.5

If your applications use ADO, make certain that you install the ADO 1.5. Refer to the Required Software section for more information.

No Oracle Support on Digital Alpha Platform

Oracle database connectivity is not supported from Digital Alpha platforms running Microsoft Transaction Server.

Oracle OCIW32.DLL Version Problem

It is important that the correct version of the OCIW32.DLL is installed on your computer. You should check the version of the DLL any time you reinstall Oracle or Microsoft Transaction Server.

DLL Name Changes in Future Releases of Oracle

Oracle sometimes changes DLL names when they release new versions of their product. Microsoft Transaction Server relies upon knowing the names of some Oracle DLLs. Currently, MTS looks for the DLL names provided with Oracle version 7.3.3. As MTS cannot predict future names of these DLLs, you may need to modify the values in the following registry key when you upgrade your Oracle installation:

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

Under this key there are two string-named values.

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

Configuring Oracle to Support a Large Number of Connections

If you want to create more than a few dozen connections to an Oracle database, you must configure the Oracle server to support additional database connections.

You may experience one or more of the following errors if you fail to do this:

- Failures on **SQLConnect** calls.
- Failures to enlist on the calling objects transaction, which may result in any of the following errors in the Oracle trace file:
  - Too many sessions
  - TNS server failed to locate the server name
  - Too many distributed transactions
- Timeouts while waiting for database locks. This is likely to occur if the configured number of locks is insufficient for the number of concurrently active transactions.

- Record collision due to locks held by in-doubt transactions.

If you experience any of these problems, increase the following Oracle server configuration parameters:

- Session count (This is typically three times the number of expected connections)
- Distributed transaction count
- Process count
- Lock count

Setting Up the MTS Sample Bank Application

The Sample Bank application is a banking services application that credits, debits, and transfers money between accounts. Running the Sample Bank application allows you to test your installation of MTS with SQL Server 6.5 as well as practice package deployment and administration. Sample Bank has components written in Visual Basic, Visual C++, and Visual J++, and is located in the \MTS\Samples subdirectory. In addition, MTS provides an Active Server Page (ASP) that calls the Sample Bank components. This sample is called Bank.asp and is located in the \MTS\Samples\asp subdirectory.

The *MTS Programmer's Guide* provides an extensive tutorial describing how to build Sample Bank components.

To run the Sample Bank application, you must:

- Select the **Custom** setup option and choose all MTS sub-components. See the Installing MTS Development Samples and Documentation topic for more information.
- Set up the DSN.
- Install the **Sample Bank** package.
- Set up the MTS Explorer to monitor the **Sample Bank** package.
- Run the Bank Client.

Setting Up the Sample Bank Application

MTS automatically configures the ODBC data source for Sample Bank during setup. Since the local machine is used by default, you must have SQL Server 6.5 installed on your local machine.

By default, the MTS DSN points to SQL Server 6.5. If you are using a database other than SQL Server 6.5, you must delete the DSN and add a new DSN called MTSSamples that points to your database.

If you want to use a SQL Server installation on another machine, use the ODBC service icon in the Control Panel to modify your data source as follows:

- In the **Data Sources** dialog box, click the **File DSN** tab, and select the MTSSamples data source.
- Click **Configure** and enter the name of the server you want to use.

Note that the Login ID and Password specified by the MTSSamples DSN are not used by Sample Bank. Sample Bank uses the "sa" account and null password. If your system administrator password is non-null or you want to specify a different login ID, you will have to modify the ODBC connection string in the Sample Bank source code.

To monitor the Sample Bank package components and transactions

1. Double-click the **Sample Bank** package icon in the right pane of the MTS Explorer.
2. Double-click the Components folder.
3. On the **View** menu, click **Status View** to display usage information for the various components in the package.
4. On the **Window** menu, click **New Window**.
5. Re-arrange the new window so the two windows do not overlap. You can stack different windows by selecting either the **Cascade** or **Tile Horizontally** options from the Window menu.
6. Click **Transaction Statistics** in the left pane of the new window.
7. On the **Action** menu, click **Scope** to clear the check mark and hide the left pane. Now transactions statistics are displayed when transactional components are used.

To run the Bank client

1. Make sure that Microsoft Distributed Transaction Coordinator (MS DTC) is running. Select **My Computer** in the left pane of the Transaction Server Explorer. Open the **Action** menu and click

Start MS DTC if that option is enabled.

2 Make sure that SQL Server is running. You can start SQL Server from Control Panel.

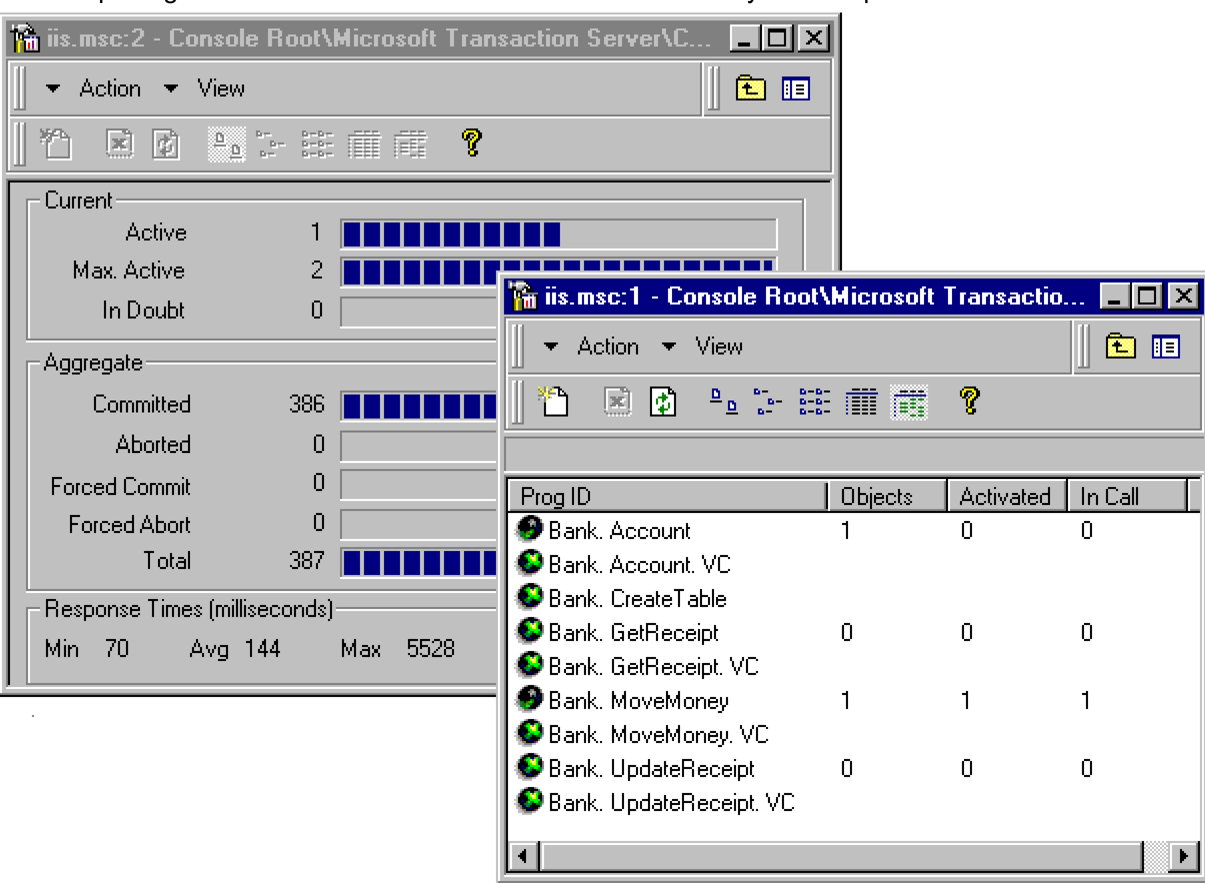
2 On the **Start** menu, point to **Programs**, point to **Microsoft Transaction Server**, point to **Samples**, and click **Bank Client**. Arrange the Bank Client window so that it does not overlap the MTS Explorer windows.

3 The form will default to credit $1 to account number 1. Click **Submit**. You should see a response with the new balance.

4 Observe the MTS Explorer windows. You will notice that the component usage and transaction statistics windows have been updated.

5 Experiment with the bank client and observe the statistics using different transaction types, servers, and iterations. The first transaction takes longer than subsequent transaction for the following reasons:

- The first transaction is creating the sample bank database tables and inserting temporary records.
- Beginning the server process consumes system resources.
- Opening database connections for the first time is a costly server operation.



Setting Up the MTS Tic-Tac-Toe Sample Application

The Tic-Tac-Toe sample application is a game that can be played with the computer or with another user on a remote MTS computer. Running the Tic-Tac-Toe application allows you to test your installation of MTS without SQL Server as well as practice package deployment and administration. The Tic-Tac-Toe sample application is located in the \MTS\Samples sub-directory.

To run the Tic-Tac-Toe sample application, you need to:

- Install Microsoft Transaction Server.
- Run the Tic-Tac-Toe client, and play against the computer or another user.

To start the Tic-Tac-Toe client

1. Run tClient.exe, located in the top-level of your MTS installation.
2. Type your name in the **Your Name is…** box, and choose if you would like to play against the computer (Deep Viper) or another user. To learn how to play against a remote user, refer to the Working with Remote MTS Computers topic.

After you start playing the game, go back to the MTS Explorer. Notice that the Tic-Tac-Toe server component icon is now spinning, indicating that it is activated and that your MTS installation is correct. If you stop the game, the icon stops spinning because the Tic-Tac-Toe client is no longer using the Tic-Tac-Toe server component.

If you click the **Status view** command on the **View** menu, you can see usage information about the Tic-Tac-Toe component.

Setting Up the MTS Administrative Sample Scripts

The administration object sample scripts demonstrate how to write scripts in an Automation-compatible language (such as Visual Basic Scripting Edition (VBScript) that automate procedures in the MTS Explorer. The sample scripts automate deployment for the Sample Bank application.

In order to run the sample scripts, you must first install the Windows Scripting Host (WSH). Note that these scripts will only run on Microsoft Windows NT and Alpha platforms.

To install the Windows Scripting Host from the Windows NT 4.0 Option Pack:

1. If you have not already installed WSH, open the **Start** menu, choose the **Microsoft Internet Information Server** option, and click **Internet Information Server Setup**.
2. Choose the **Add/Remove** option from the setup program.
3. Find the Windows Scripting Host option in the list of components, and select the checkbox.
4. Click **Finish**. The Windows NT 4.0 Option Pack setup program will install WSH on your computer.

Once you have WSH installed, you can use the administration object sample scripts. The following sample scripts are located in your \MTS\Samples\WSH sub-directory:

- InstDLL.vbs

  Deletes existing versions of Sample Bank, creates a new package named Sample Bank, installs components from the Sample Bank Visual Basic, Visual C++, and Visual J++ DLLs into the new package, changes transaction attributes, and adds a new role. You must modify the file path before running the script.

  Note this script requires that the computer registry contain all components' programmatic identifiers (progIDs). Therefore, you must use the Visual Studio 97 ActiveX Wizard to register your Java components before you run this script.
- InstPak.vbs

  Installs the Sample Bank package into the MTS run-time environment. You must modify the file path before running the script.
- Uninst.vbs

  Uninstalls the Sample Bank package from the MTS run-time environment. You must modify the file path before running the script.
- InstDllCLI.vbs

  Allows you to run the InstDll.vbs script from a console window and enter the file path as a parameter to delete existing versions of Sample Bank, create a new package named Sample Bank, install components from the Sample Bank Visual Basic, Visual C++, and Visual J++ DLLs into the new package, change transaction attributes, and add a new role.
- InstPakCLI.vbs

  Allows you to run the InstPak.vbs script from a console window and enter the file path as a parameter to install the Sample Bank package into the MTS run-time environment.

For more information about the scriptable administration objects and the sample scripts, refer to the Automating MTS Administration topic in the *MTS Administrator's Guide*.

Limitations of the Administrative Sample Scripts

- Java components must be registered using the Visual Studio 97 ActiveX Wizard before running the InstDLL.vbs script.
- Installing Java components using the sample scripts is only supported on an Alpha or i386 computer. Alpha users should remove the Java component importing code from InstDLL.vbs before running the script.

Getting Assistance While You Work with MTS

You can get assistance while you work by:

- Consulting the documentation.
- Visiting the Microsoft Transaction Server home page at http://www.microsoft.com/transaction/.
- Contacting product support services.

Consulting the Documentation

The MTS documentation contains conceptual, task-oriented, and reference information about MTS features. You can also access Help by pressing F1 to get context-sensitive Help while using the MTS Explorer and programming language keywords. *Context-sensitive* means that you can get Help directly without having to go through the **Help** menu—you just press F1.

See Also MTS Documentation Roadmap

Product Support Services

Microsoft offers a variety of support options to help you get the most from MTS.

If you have a question about the product, first look in Help. If you cannot find the answer, contact Microsoft Product Support Services.

Support services are available both within the United States and through subsidiary offices worldwide.

Roadmap to the MTS Administrator's Guide

The MTS Administrator's Guide describes how and when to use the Microsoft® Transaction Server Explorer to create, install, distribute, and maintain packages. This guide contains information for the following users:

- System administrators
- Web administrators
- Application developers

Developers can refer to the procedures when creating, deploying, and distributing MTS applications. System and Web administrators can use the task-oriented procedures for the MTS Explorer to deploy, administer and maintain MTS applications.

The following sections describe what it means to deploy and administer packages with the MTS Explorer, and provide links to more detailed procedural topics:

What Does Creating an MTS Package Mean?

What Does Distributing an MTS Package Mean?

What Does Installing an MTS Package Mean?

What Does Maintaining an MTS Package Mean?

What Does Managing an MTS Transaction Mean?

What Does Automating MTS Administration Mean?

MTS Explorer on Windows 95

You can manage your MTS packages using the MTS Explorer on the Windows® 95 operating system. However, MTS administration on Windows 95 has the following limitations:

- You can use a Windows NT computer to remotely administer a Windows 95 computer if you run the Remote Registry service on the Windows 95 computer. The Remote Registry service allows you to change registry entries for a remote Windows 95 computer (given the appropriate permissions).

  To obtain the Remote Registry service, go to the \Admin\Nettols\Remotereg sub-directory on the Windows 95 CD. Review the Regserv.txt file for instructions on installing the Remote Registry service, and then run the Remote Registry setup program (Regserv.exe).
- You cannot remotely administer MTS running on Windows 95 from another Windows 95 computer nor a Windows NT computer.
- The left tree pane in the MTS Explorer does not appear. To navigate, double-click icons to move down the hierarchy and then click the **Up one level** toolbar button to move up the hierarchy.
- Because the application executable utility is not supported on Windows 95, you cannot use the MTS Explorer to generate executables. For more information on the application executable utility, see the Generating MTS Executables topic.
- Windows 95 administration does not support MTS security properties or roles. Therefore, you will not be able to view the Roles, Role Membership, or Users folders in the Explorer.
- Components running on Windows 95 cannot be accessed remotely from a client on another computer.
- Because Windows 95 does not have a system event log, event log entries are written to an HTML file named Transaction Server.html, which is located in the \MTSLogs subdirectory in the Windows directory. You can use this HTML file to monitor any significant occurrence in the system or a program.

See Also

Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server

What Does Creating an MTS Package Mean?

Creating packages is the final step in the development process. Package developers and advanced system and Web administrators use the MTS Explorer to create and deploy packages. You use the Explorer to implement package and component configuration that is determined during development of the application.

For more information about how to design and build MTS applications, see the Building MTS Applications section of the *MTS Programmer's Guide*.

Procedures:

Creating an Empty MTS package

Adding a Component to an MTS Package

Importing a Component into an MTS Package

Removing a Component from an MTS Package

Building an MTS Package for Export

Setting MTS Package Properties

Setting MTS Activation Properties

Setting MTS Transaction Properties

Setting MTS Authentication Levels

Locking Your MTS Package

See Also

Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server , What Does Distributing an MTS Package Mean?, What Does Installing an MTS Package Mean?, What Does Maintaining an MTS Package Mean?, What Does Managing an MTS Transaction Mean?, What Does Automating Administration for Packages Mean?

What Does Distributing an MTS Package Mean?

After the components of an MTS application have been built and packaged, you then distribute the application to clients.You distribute applications in the following ways:

- Push components from your server computer to a system or Web site administrator's server computer using the MTS Explorer. In this case, it is required that both server computers be running MTS.
- Use the application executable utility in the MTS Explorer to automatically generate application executables that reference a remote server. The client application does not have be running MTS.

It is recommended that you use the MTS Explorer application executable utility to distribute your server package to clients who may or may not have MTS on their computers. The application executable automatically configures a client computer to access components running on a remote MTS server. You can also configure remote components manually using the MTS Explorer.

Although using the application executable utility does not require programming knowledge, MTS application distributors should be thoroughly familiar with the implications of packaging and shipping client and server applications. For example, improper packaging of your application may result in providing clients with server application code in the client executable.

Procedures:

Working with Remote MTS Computers

Exporting MTS packages

Generating MTS Executables

See Also

Quick Tour of MTS, Getting Started with Microsoft Transaction Server , What Does Creating an MTS Package Mean?, What Does Installing an MTS Package Mean?, What Does Maintaining an MTS Package Mean?, What Does Managing an MTS Transaction Mean?, What Does Automating MTS Administration Mean?

What Does Installing an MTS Package Mean?

After you build a package, you install and deploy it, which requires familiarity with the package and components properties. For example, after installing a package, a system or Web administrator must map Windows NT users to the roles associated with the package. The system or Web site administrator should thoroughly understand role-based declarative security and the roles associated with an application before mapping application users and groups to roles.

Procedures:

Installing Pre-built MTS packages

Upgrading MTS Packages

Enabling MTS Package Security

Setting MTS Package Identity

Adding a New MTS Role

Mapping MTS Roles to Users and Groups

See Also

Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server , What Does Creating an MTS Package Mean?, What Does Distributing an MTS Package Mean?, What Does Maintaining an MTS Package Mean?, What Does Managing an MTS Transaction Mean?, What Does Automating MTS Administration Mean?

What Does Maintaining an MTS Package Mean?

You can use the MTS Explorer to maintain MTS applications by monitoring the status of installed packages and re-configuring component and package properties if applicable. This section describes how you can re-configure packages that are already installed and deployed.

Procedures:

Monitoring Status and Properties in the MTS Explorer

Using Property Sheets in the MTS Explorer

Managing Users for MTS Roles

Using MTS Replication

See Also

Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server , What Does Creating an MTS Package Mean?, What Does Distributing an MTS Package Mean?, What Does Installing an MTS Package Mean?, What Does Managing an MTS Transaction Mean?, What Does Automating MTS Administration Mean?

What Does Managing MTS Transactions Mean?

As an administrator, you should understand how distributed transactions work in order to understand the context in which you are managing transactions in the MTS Explorer. This section discusses transactions and transaction states, and how monitor and manage the transactions that you are administering.

Procedures:

Understanding MTS Transactions

Managing MS DTC

Monitoring MTS Transactions

Monitoring MTS Transactions on Windows 95

Understanding MTS Transaction States

Resolving MTS Transactions

See Also

Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server , What Does Creating an MTS Package Mean?, What Does Distributing an MTS Package Mean?, What Does Installing an MTS Package Mean?, What Does Maintaining an MTS Package Mean?, What Does Automating MTS Administration Mean?

What Does Automating MTS Administration Mean?

Scriptable administration enables you to automate MTS Explorer configuration of packages and components. If you know a scripting language such as Microsoft Visual Basic, you can use the scripting objects to automate administration tasks in the Explorer. For example, tool developers can also use the scriptable objects to create MTS Explorer add-ins such as an object that automatically configures remote clients.

Note Using the scriptable objects requires a working knowledge of an Automation scripting language.

You must install MTS development samples and documentation in order to obtain the administrative sample scripts and *MTS Administrative Reference*, which contains Microsoft Visual Basic and Microsoft Visual C++® API reference pages and sample code for the scriptable objects. The administrative sample scripts are written in Visual Basic Script and take advantage of the Windows Scripting Host, which can be installed using the the Windows NT 4.0 Option Pack setup program. See the Setting Up the MTS Administrative Sample Scripts topic for more information about using the MTS administrative scripts.

Procedures:

MTS Administration Objects

Visual Basic Script Sample for Automating MTS Administration

Visual Basic Sample Application for Automating MTS Administration

Automating MTS Administration with Visual Basic

Automating Advanced MTS Administration with Visual Basic

See Also

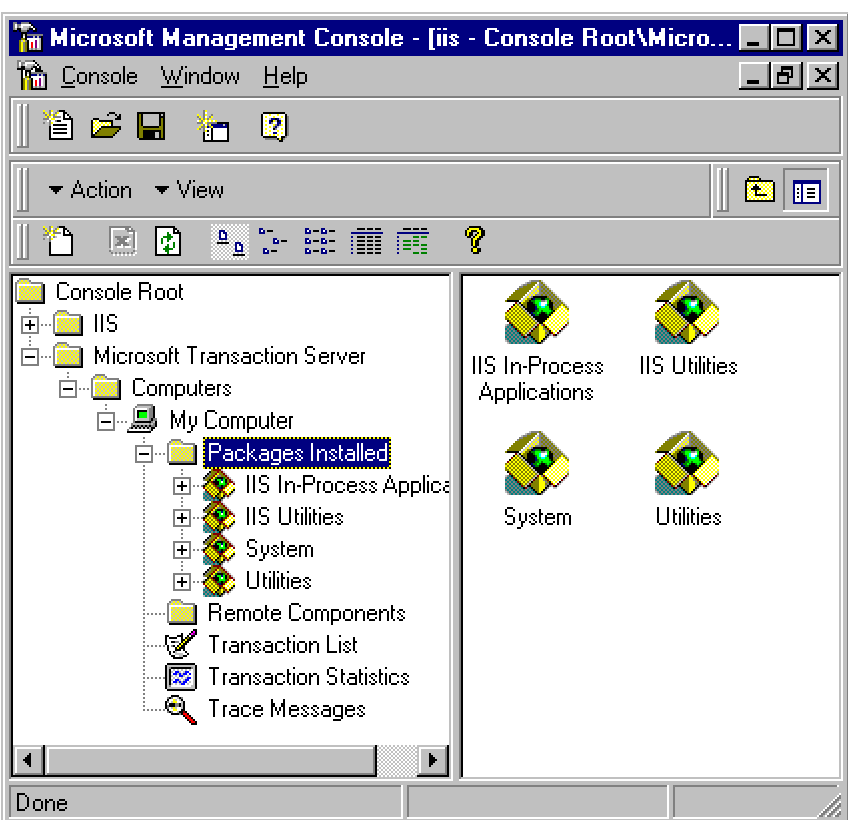
Quick Tour of Microsoft Transaction Server, Getting Started with Microsoft Transaction Server , What Does Creating an MTS Package Mean?, What Does Distributing an MTS Package Mean?, What Does Installing an MTS Package Mean?, What Does Maintaining an MTS Package Mean?, What Does Managing an MTS Transaction Mean?

MTS Explorer Hierarchy

The MTS Explorer is the visual tool used to manage MTS packages and components executing in the MTS run-time environment. You use the MTS Explorer to perform tasks ranging from installing components into packages to monitoring the status of transactions.

The MTS Explorer is a snap-in hosted by the Microsoft Management Console (MMC) on Windows NT and an executable file (mtxpd.exe) on Windows 95. The Explorer contains the following folders in the left tree view hierarchy:

- **Computers**
  Contains the computers managed from this server. The local computer is named My Computer.
- **Packages Installed**
  Contains the packages installed on a given computer.
- **Remote Components**
  Contains the components on remote computers used by packages on a given computer.
- **Transaction List**
  Displays the current transactions in which an MTS application participates.
- **Transaction Statistics**
  Displays statistics on the transactions in which the local computer participates.
- **Trace Messages**
  Lists current trace messages issued by the Microsoft Distributed Transaction Coordinator (MS DTC).



Note The MTS Explorer on Windows 95 has a single pane. To navigate in the MTS Explorer hierarchy on Windows 95, double-click the icons to move down the hierarchy and then click the **Up One Level** toolbar button to move up the hierarchy.

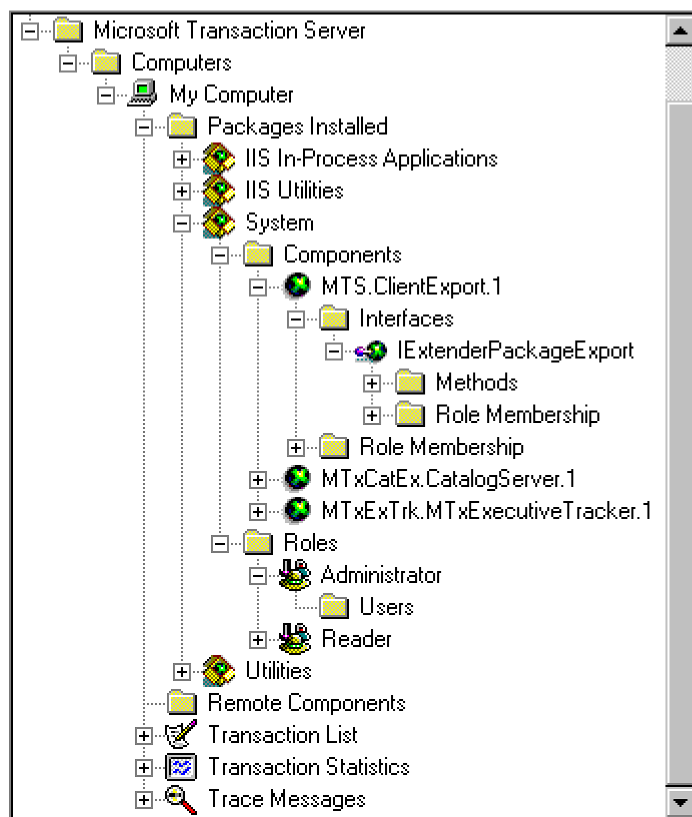
In addition, you cannot use the Remote Components folder or the application executable utility on Windows 95.

See the *MTS Explorer on Windows 95* section of the Road Map to the MTS Administrator's Guide topic for limitations of the MTS Explorer on Windows 95.

Remote computers administered by the MTS Explorer list the Packages Installed, Remote Components, Transaction List, Transaction Statistics, and Trace Messages windows in the hierarchy below the computer icon.

Each package installed in the MTS Explorer contains the following sub-folders:

- **Components**
  Contains the components installed in the selected package
- **Roles**
  Contains the roles available within a package.
- **Role Membership**
  Contains the roles assigned to the selected component or component interface.
- **Users**
  Contains the users mapped to the role for the package.
- **Interfaces**
  Contains the interface(s) for the selected component.
- **Methods**
  Contains the methods for the selected interface.



See Also

Quick Tour of Microsoft Transaction Server

Computers Folder

The Computers folder contains My Computer and other computers that you have added to your Computers folder. By default, My Computer corresponds to the local computer on which MTS is installed.

You can add a computer to the Computers folder by doing one of the following:

- Right-clicking the Computers folder and choose **New** and then **Computer**.
- Selecting the Computers folder and clicking on the **Create new object** icon in the right pane toolbar
- Selecting the Computers folder, opening the **Action** menu in the left pane of the Explorer, and choosing **New**.

In the **Add Computer** dialog box, type the name of the server that you would like to administer from your computer. The new server is added to the Computers folder below the My Computer icon in the MTS Explorer hierarchy.

See Also

My Computer, Computer Properties

My Computer

My Computer corresponds to the local computer on which MTS is installed.

See the following topics for procedures can be done at the Computer folder level:

Using MTS Replication

Managing MS DTC

All Computer folders contain the following folders:

Packages Installed

Contains the user-installed and system packages managed on this computer.

Remote Components

Contains the components on remote computers invoked by packages on the local computer.

Transaction List

Lists the current transactions being managed on this computer.

Transaction Statistics

Displays statistics on the transactions in which a computer participates.

Trace Messages

Lists current trace messages issued by the Microsoft Distributed Transaction Coordinator (MS DTC).

As with every computer in the Computers folder, the properties for My Computer can be configured using the following property sheets:

- General
- Options
- Advanced

See Also

Computers Folder

Packages Installed Folder

The Packages Installed folder lists all the packages that you have added to a given computer. You can perform the following functions at the package level:

Creating an Empty MTS Package
Installing Pre-built MTS Packages
Upgrading MTS Packages
Setting MTS Package Properties
Setting MTS Activation Properties
Setting MTS Package Identity
Enabling MTS Package Security
Setting MTS Transaction Properties
Locking Your MTS Package
Exporting MTS Packages
Monitoring Status and Properties in the MTS Explorer
Using Property Sheets in the MTS Explorer

The Packages Installed folder also contains the following system packages:

System

Utilities

Note These are internal MTS packages, and generally should not be altered or configured by users. However, you may have to configure an internal MTS package in order to extend or restrict privileges for your MTS server. For example, in order to set up administrative privileges for a user on an MTS server, you add the user to the Administrator role for the System package.

You can install any number of packages in the Packages Installed folder. Each installed package contains the following subfolders.

- Components
  Contains the components installed in the selected package
- Roles
  Contains the roles assigned to the selected package.

Package properties can be configured using the following property sheets:

- General
- Security
- Advanced
- Identity
- Activation

If you are using MTS with Internet Information Server (IIS) version 4.0, the Packages Installed folder also contains the following IIS-specific system packages:

- IIS In-Process Applications
  The IIS In-Process Applications folder contains the components for each Internet Information Server (IIS) application running in the IIS process. An IIS application can run in the IIS process or in a separate application process. If an IIS application is running in the IIS process, the IIS application will appear as a component in the IIS In-Process Applications folder. If the IIS application is running in an individual application process, the IIS application will appear as a separate package in the MTS Explorer hierarchy.

- IIS Utilities

  The IIS Utilities Folder contains the ObjectContext component required to enable transactions in Active Server Pages (ASPs). For more information about transactional ASPs, refer to the Internet Information Server (IIS) documentation.

  Microsoft Transaction Server uses the ObjectContext component for internal functions. You can view but not set this component's properties.

Utilities Package

The Utilities package includes two components, **TransactionContext** and **TransactionContextEx**. You can use **TransactionContext/TransactionContextEx** in your base clients to compose the work of one or more Microsoft Transaction Server objects into an atomic transaction, and to abort or commit the transaction.

See Also

Managing MTS Transactions

System Package

The System package includes components that cannot be modified. Microsoft Transaction Server uses these components for internal functions. You can view but not set System component properties.

In order for a user to be able to delete and modify packages managed by the MTS Explorer, that user must be mapped to the Administrator role on the System package. If MTS is installed on a server whose role is a primary or backup domain controller, a user must be a domain administrator in order to manage packages in the MTS Explorer.

See Also

Configuring Your MTS Server, Enabling MTS Package Security, Mapping MTS Roles to Users and Groups, Users Folder

Components Folder

The Components folder contains the components in a selected package.

See the following topics for procedures at the component level:

Adding a Component to an MTS Package
Importing a Component into an MTS Package
Removing a Component from a Package
Monitoring Status and Properties in the MTS Explorer
Using Property Sheets in the MTS Explorer
Setting MTS Transaction Properties
Configuring MTS Roles for Declarative Security

The Components folder contains the following subfolders:

- Interfaces
  Contains the interface(s) associated with the selected component
- Role Membership (Components)
  Contains the roles and users associated with roles for the selected component

Component properties can be configured using the following property sheets:

- General
- Transaction
- Security

See Also

Setting MTS Activation Properties

Roles Folder

The Roles folder contains the roles assigned for a selected package. MTS allows you to define roles that determine user access for a package, component, or interface. Any role you add to the Role folder for the selected package you can also add to the Role Membership folder for a component or an interface in that package.

You can perform the following functions at the Roles level:

Mapping MTS Roles to Users and Groups
Enabling MTS Package Security
Adding a New MTS Role

The Roles folder has one subfolder:

- Users

Role properties can be configured using the following property sheet:

- General

See Also

Configuring Your MTS Server, System Package

Interfaces Folder

The Interfaces folder contains the interfaces defined for a selected component.

The Interfaces Folder contains two subfolders:

- Methods
- Role Membership

Interface properties can be viewed using the following property sheets:

- General
- Proxy

Note that you cannot configure interface properties aside from providing a description of the interface on the General property sheet.

See Also

Enabling MTS Package Security

Methods Folder

The Methods folder contains the methods defined in a selected interface.

Method properties have a single property sheet:

- General

You cannot configure a method other than adding a description of the method in the General property sheet.

See Also

InterfacesFolder

Role Membership Folder

The Role Membership folder contains the roles that you associated with a component or interface. When you add these roles the Role Membership folder from a package's Roles folder, you control who can access an interface or component.

You set the properties for Roles Membership at the Role folder level. For example, the description that you enter for a role for the package will be displayed for that role at the Role Membership level.

See the Enabling MTS Package Security topic to learn about declarative security.

See Also

Interfaces Folder

Users Folder

The Users folder contains the Windows NT users or groups that you associate with a role. Each user represents a Windows NT user account that you add to the Roles folder of a package. You can control access to packages, components, and interfaces by adding Windows NT user accounts or groups to the Roles folder.

You can perform the following functions at the Users level:

Adding a new user to a role
Removing a user from a role

There are no property sheets associated with the Users folder.

See Also

Enabling MTS Package Security, Mapping MTS Roles to Users and Groups

Remote Components Folder

The Remote Components folder contains the components that are registered locally on this computer to run remotely on another computer. Using the Remote Components folder requires that you have MTS installed on the client machines that you want to configure. If you want to configure remote computers manually using the Explorer, add the components that will be accessed by remote computers to the Remote Components folder.

Note that before you can configure remote components, you must add to your Computers folder any additional servers that will run remote components.

See the following topics to learn more about working with remote computers that are running MTS:

Exporting MTS Packages

Working with Remote MTS Computers

Note Components running on Windows 95 cannot be accessed remotely from a client on another computer. See the *MTS Explorer on Windows 95* section of Road Map to the MTS Administrator's Guide.

See Also

Generating MTS Executables

Transaction List

The **Transaction List** window displays the current transactions in which this computer participates, including:

- Transactions whose status is in-doubt.
- Transactions that have remained in the same state for the period of time specified on the **Advanced** tab on the Computer property sheets.

See the Transaction Icons topic for a description of the icons displayed in Transaction List

See the following topics for an overview of managing transactions:

- Understanding MTS Transactions
- Understanding MTS Transaction States

You can use the **Transaction List window** for to do the following tasks:

- Monitoring MTS Transactions
- Monitoring MTS Transactions with Windows 95
- Resolving MTS Transactions

You can also view properties of a transaction by right-clicking on the selected transaction and clicking the **Properties** option.

See Also

Transaction Icons, Managing MS DTC

Transaction Statistics

The **Transaction Statistic** window displays statistics on the transactions in which a computer participates. Some of the statistics are cumulative; others reflect current performance.

See the following topics for an overview of managing transactions:

- Understanding MTS Transactions
- Understanding MTS Transaction States

You can use the **Transaction Statistics window** for the following tasks:

- Monitoring MTS Transactions
- Monitoring MTS Transactions with Windows 95
- Resolving MTS Transactions

Note that you are using the MTS Explorer on Windows NT, you can only open one Transaction Statistics window for a given server.

Current

- **Active** — The current number of transactions that have not yet completed the two-phase commit protocol.
- **Max. Active** — The highest number of active transactions at any time during the current Microsoft Distributed Transaction Coordinator (MS DTC) session.
- **In Doubt** — The current number of transactions that are unable to commit because of a communication failure between the local database server and the commit coordinator.

Aggregate

- **Committed** — The cumulative total of committed transactions. This number does not include forced (manually resolved) commits.
- **Aborted** — The cumulative total of aborted transactions. This number does not include forced aborts.
- **Forced Commit** — The cumulative total of manually committed transactions.
- **Forced Abort** — The cumulative total of transactions that were manually aborted.
- **Total** — The cumulative total of all transactions.

Response Times

This group shows the minimum, average, and maximum transaction response times in milliseconds. The response time is the duration of a transaction from the point when it began to the point when it was committed.

MS DTC Started/MS DTC Stopped

This group shows the date and time that the current MS DTC session started. The date and time started will not appear unless MS DTC is started. The group will also show that MS DTC is stopped.

Some statistics are cumulative; others reflect current performance.

When you stop the MS DTC service, the values of all cumulative statistics are reset to zero.

See Also

Managing MS DTC

Trace Messages

The **Trace Messages** window lists current trace messages issued by the Microsoft Distributed Transaction Coordinator (MS DTC). Tracing allows you to view the current status of various MS DTC activities, such as start up and shut down, and to trace potential problems by viewing additional debugging information.

For an overview of managing transactions, see the following topics:

- Understanding MTS Transactions
- Understanding MTS Transaction States

You can use the **Trace Messages** window for the following tasks:

- Monitoring MTS Transactions
- Monitoring MTS Transactions with Windows 95

You can use the **Trace** slider on the **Advanced** tab of a computer's property sheet to specify the level of tracing that is displayed in this window.

Severity

Icon	Description
	Errors Something has happened that requires restarting MS DTC. For example, a corrupt log file is detected.
	Warnings Something could go wrong soon.
	Information Information is provided about infrequent events, such as start up and shut down.
	Trace Debugging information is provided about events such as new client connections or resource manager enlistments.

Source

Displays the source of the trace message:

- **SVC** — The MS DTC Service is the source of the trace message.
- **LOG** — The MS DTC log is the source of the trace message.
- **CM** — The MS DTC network connection manager is the source of the trace message.

The tracer and the Windows NT event log tag each message with its source.

Message

Displays the message.

See Also

Managing MS DTC

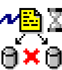
Transaction Icons

The following icons are displayed in the Transaction List:

Icon | **Description**

Active

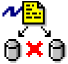
The transaction has been started



Aborting

The transaction is aborting. MS DTC is notifying all participants that the transaction must abort.

It is not possible to change the transaction outcome at this point.



Aborted

The transaction has aborted. All participants have been notified. Once a transaction has aborted, it is immediately removed from the list of transactions in the MS DTC Transactions window.

It is not possible to change the transaction outcome at this point.

Preparing

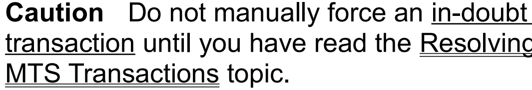
The client application has issued a commit request. MS DTC is collecting prepare responses from all participants.

Prepared

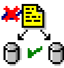
All participants have responded yes to prepare.

In Doubt

The transaction is prepared, is coordinated by a different MS DTC, and the coordinating MS DTC is inaccessible. The system administrator can force the transaction to commit or abort by right-clicking in the Transactions window and choosing the **Resolve/Commit** or **Resolve/Abort** command. Once an outcome is forced, the transaction is designated as forced commit or forced abort.

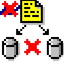


Caution Do not manually force an in-doubt transaction until you have read the Resolving MTS Transactions topic.



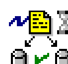
Forced Commit

The administrator forced the in-doubt transaction to commit (see the Resolving MTS Transactions topic).



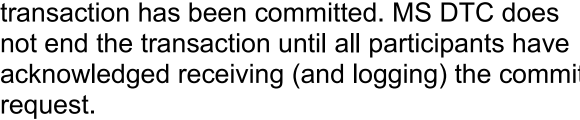
Forced Abort

The administrator forced the in-doubt transaction to abort (see the Resolving MTS Transactions topic).



Committing

The transaction has prepared successfully and MS DTC is notifying participants that the



transaction has been committed. MS DTC does not end the transaction until all participants have acknowledged receiving (and logging) the commit request.

It is not possible to change the transaction outcome at this point.

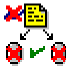
Cannot Notify Aborted

MS DTC has notified all connected participants that the transaction has aborted. The only participants not notified are those that are currently inaccessible.

This transaction state occurs when MS DTC must inform any resource manager (such as an IBM LU 6.2 system) that a transaction has aborted but is unable to do so because the connection to the IBM system is down.

The system administrator can force MS DTC to forget the transaction by right-clicking in the Transactions window and choosing the **Resolve/Forget** command.

Caution Do not manually forget a transaction until you have read the Resolving MTS Transactions topic.

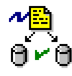


Cannot Notify Committed

MS DTC has notified all connected participants that the transaction has committed. The only participants not notified are those that are currently inaccessible.

The system administrator can force MS DTC to forget the transaction by right-clicking in the Transactions window and choosing the **Resolve/Forget** command.

Caution Do not manually forget a transaction until you have read the Resolving MTS Transactions topic.



Committed

The transaction has committed and all participants have been notified. Once a transaction commits, it is immediately removed from the list of transactions in the MS DTC Transactions window.

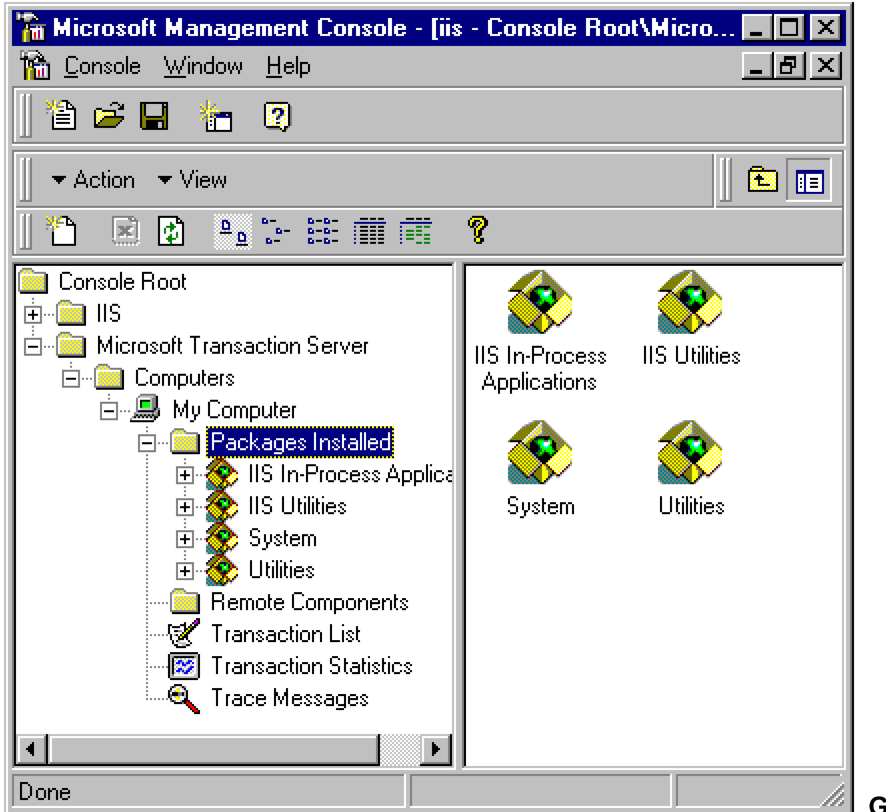
It is not possible to change the transaction outcome at this point.

See Also

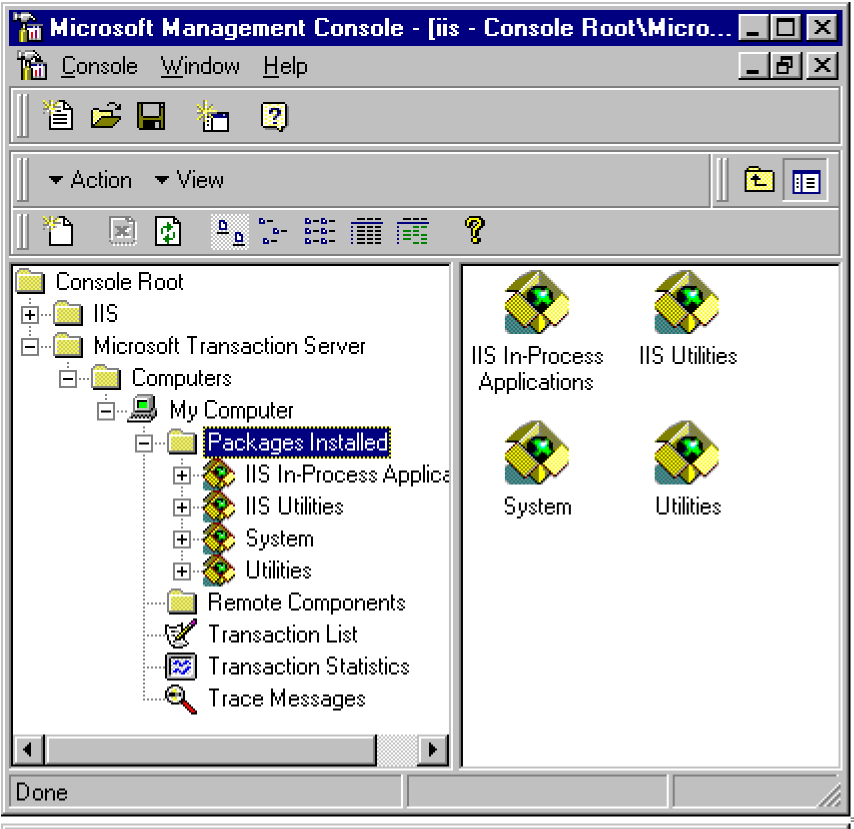
Managing MS DTC

Computer Properties

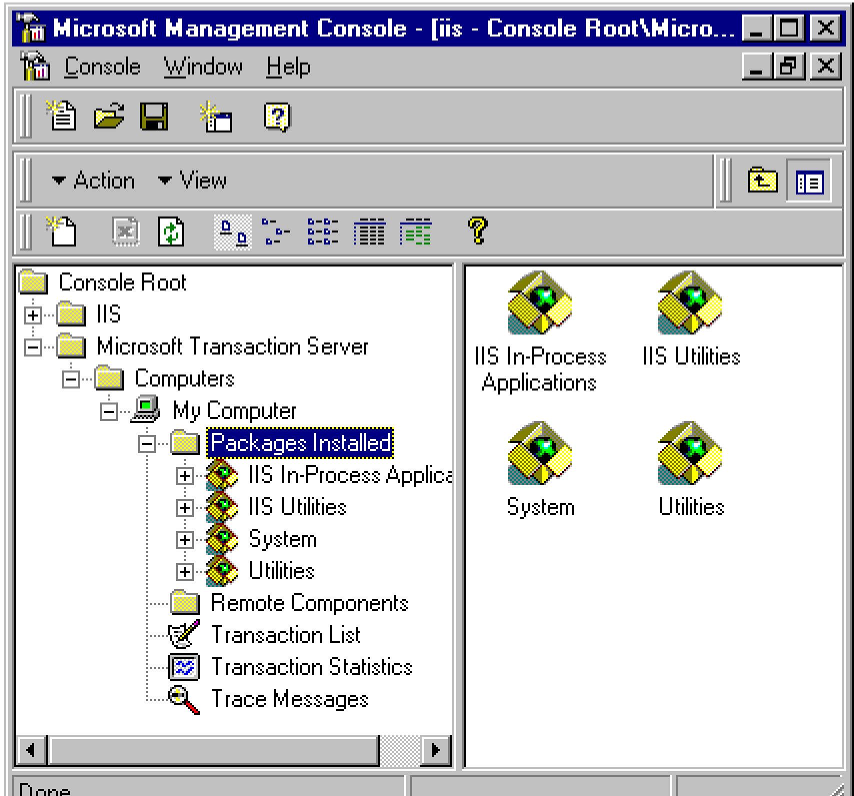
Computer properties determine general information about the computer and control how Microsoft Transaction Server updates the computer.



General



Options



Advanced

See Also

Computers Folder

General Tab (Computer)

The **General** tab defines the computer's name and description.

Name

Displays the name of the computer.

Description

Displays a description of the computer. You can type a description to help you identify and manage the computer.

See Also

Computers Folder

Options Tab (Computer)

The **Options** tab is used to set the computer's transaction timeout property and replication information.

The transaction timeout value is measured in seconds and indicates the maximum time period that transactions started on this computer can remain active. Transactions that remain active beyond the specified time are automatically aborted by the system. The default value is 60 seconds. You can disable transaction timeouts by specifying the value 0; this setting is particularly useful when debugging MTS applications.

Use the Replication section of the **Options** tab to provide replication information for your MTS computer. In the **Replication share** box, enter the name of your Microsoft Cluster Server (MSCS) virtual server name to ensure failover support. You cannot replicate an MTS catalog on a Windows 95 computer.

You can also specify a computer that you want your client executables to access. Enter the name of the physical server for your clients to access in the **Remote server name** box before you generate the application executable. If this string is blank or empty, the physical computer name of the exporting computer will be used. If you put the name of the remote server as the string, the application executable generated by the MTS Explorer will point to that remote server name.

See Also

Using MTS Replication, Generating MTS Executables, Computers Folder

Advanced Tab (Computer)

The **Advanced** tab is used to configure properties for the Microsoft Distributed Transaction Coordinator (DTC). These settings only apply to the Transaction List, Transaction Statistics, and Trace Messages windows.

Note You will not be able to access the Help documentation from the **Advanced** tab of the My Computer property sheet.

View

- **Display Refresh** — Ranges from **Infrequently** to **Frequently** with a slider bar. **Infrequently** means the transaction windows are updated every 20 seconds, whereas **Frequently** means they're updated every 1 second. More frequent updates increase administrative overhead on the running transactions but also provide more current information.
- **Transactions Shown** — Ranges from **Very Old** to **New + Old** with a slider bar. Selecting **Very Old** displays transactions only after they have been active for 5 minutes, while selecting **New + Old** displays transactions that have been active for 1 second or more.
- **Trace** — Ranges from **Less (faster MS DTC)** to **More (slower MS DTC)** with a slider bar. The settings are as follows:
  - Send no traces.
  - Send only error traces.
  - Send error and warning traces.
  - Send error, warning, and informational traces (default).
  - Send all traces.

Log Properties

- **Location** — Where to store the log file.
- **Capacity** — The maximum size of the log file.

Reset Log

Used to update the log file after any changes have been made.

Caution	Do not reset the MS DTC log file while there are unresolved transactions.

See the following topics for an overview of managing transactions:

- Understanding MTS Transactions
- Understanding MTS Transaction States

See the following topics to learn how to use the **Transaction Statistics**, **Trace Messages**, and **Transaction List window**s in the Explorer:

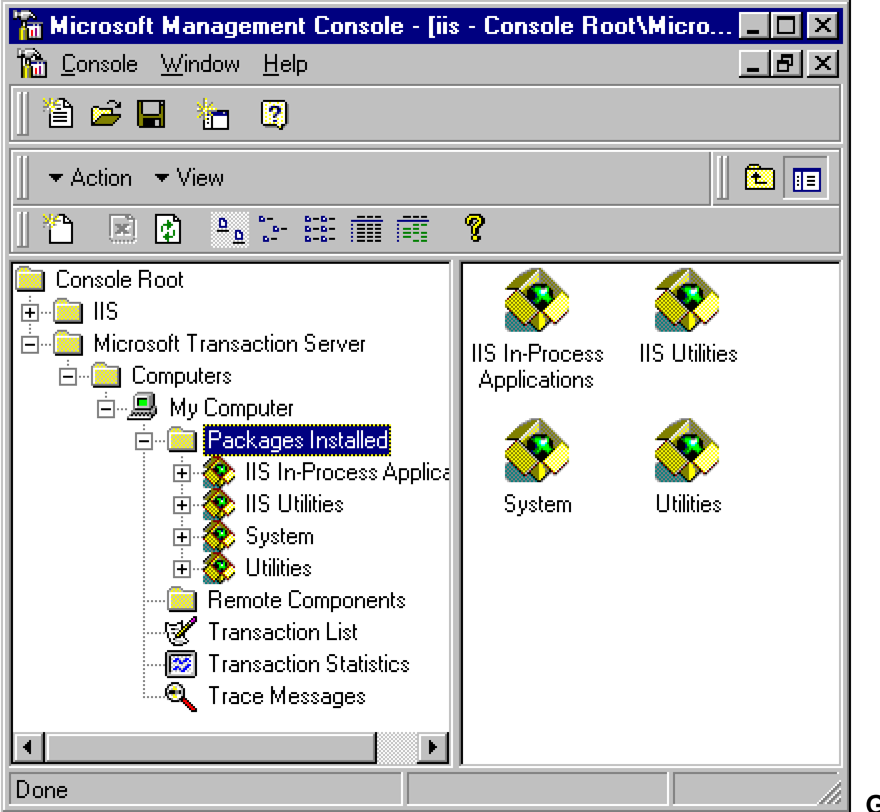
- Monitoring MTS Transactions
- Monitoring MTS Transactions with Windows 95

See Also

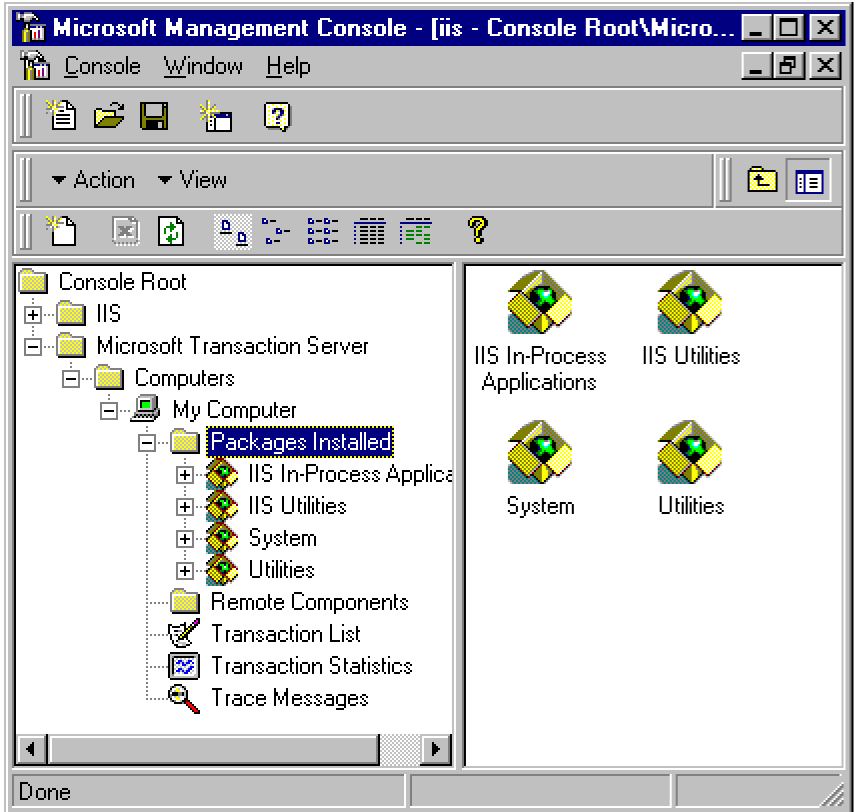
Managing MS DTC

Package Properties

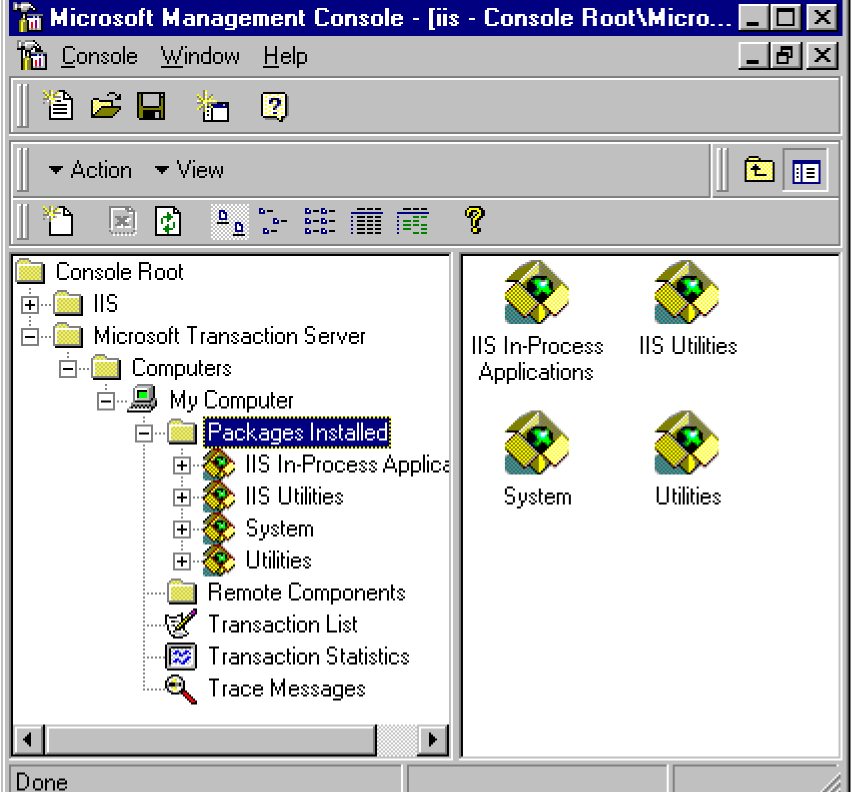
Package properties control how a package is accessed.



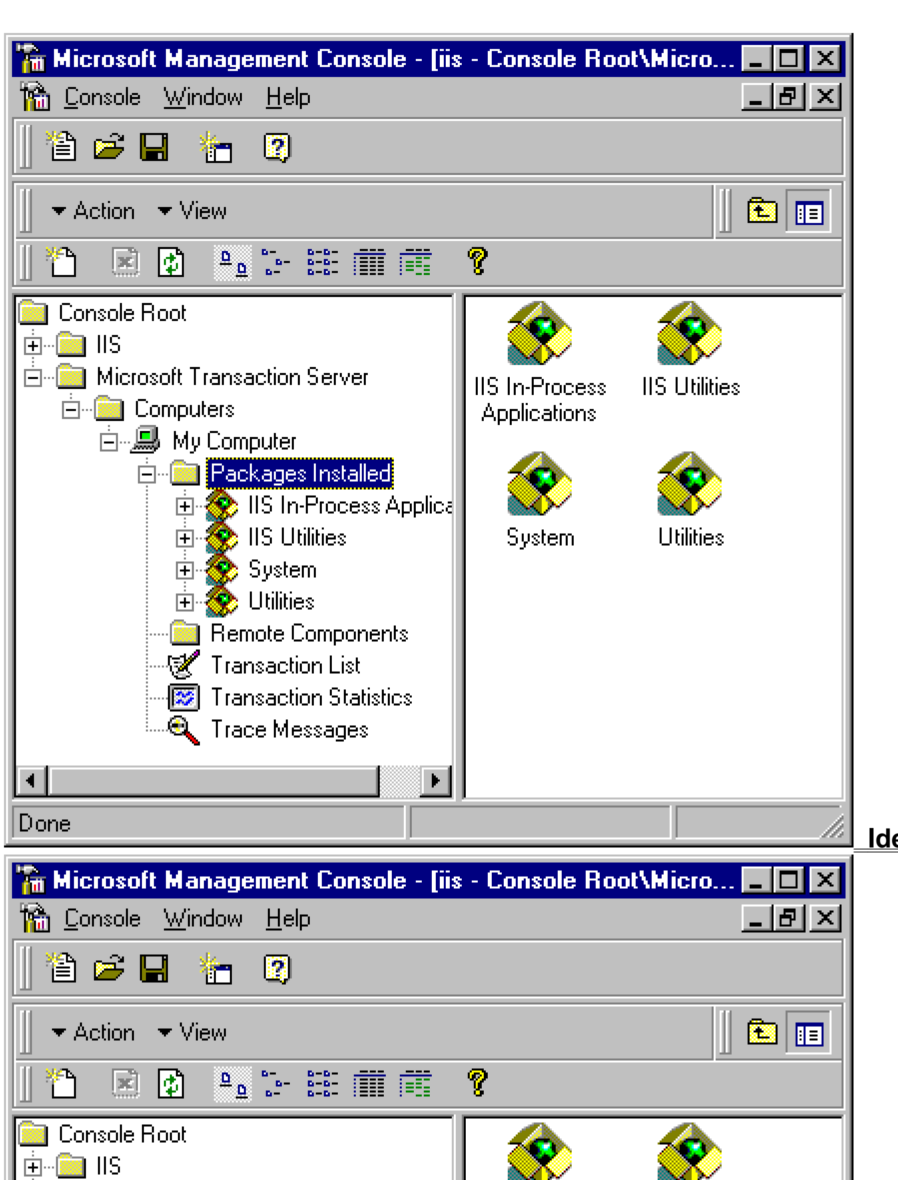
General



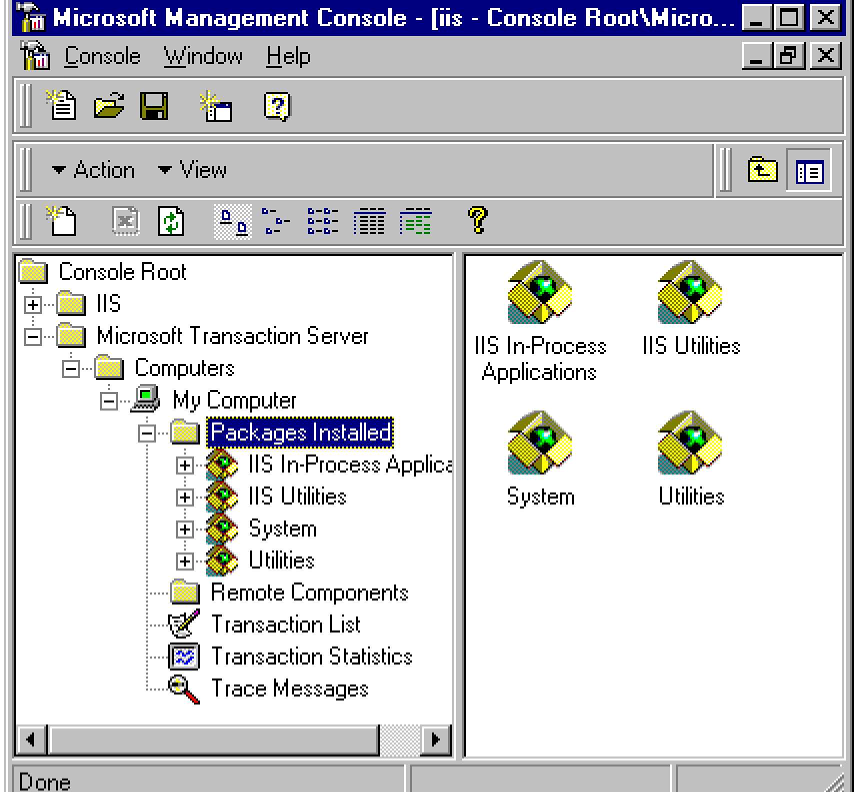
Security



Advanced



Identity



Activation

Important You cannot modify **Security** or **Identity** properties (or shut down a package) using the property sheets for Library packages.

On Windows 95 computers, the package property sheets do not include the **Security** tab, as role-based security is only supported on Windows NT.

See Also

Packages Installed Folder

General Tab (Package)

The **General** tab displays general information about the selected package.

Name

Displays the name of the package.

Description

Displays a description of the package. You can type a description to help you identify and manage the package.

Package ID

Displays the package identification number, a unique number that is generated when you create the package. You can use the Package ID to identify particular versions of a package on a computer.

See Also

Packages Installed Folder, Package Properties

Security Tab (Package)

The **Security** tab displays security information about the selected package.

Enable authorization checking

If selected, Microsoft Transaction Server checks the security credentials of any client that calls the package. Authorization checking is enabled by default.

Authentication level for calls

The level of authentication for clients calling the package.

See Also

Packages Installed Folder, Package Properties, Setting MTS Authentication Properties, Enabling MTS Package Security

Advanced Tab (Package)

The **Advanced** tab determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.

If you want the package to shut down automatically after a certain period of inactivity, you can use the **Shut down after being idle for** selection to set when the server process should be shut down.

If you want the server process always to be available, you should select the **Leave running when idle** option.

You can use the **Shut Down Server Processes** command from the **Tools** menu to shut down all server processes running on the selected computer.

See Also

Packages Installed Folder

Identity Tab (Package)

The **Identity** tab is used to set the user identity for all components running in a given package. The default value is **Interactive User**, which is the user who logged on to the Windows NT server account. If you want to select another user, you can select the **This user** option and specify an account name and password.

Important If you specify another user and password, Microsoft Transaction Server does not validate the password when it is specified. Running a package with an invalid password results in a run-time error and a message in the event log.

To set the **This user** option to a user or group, you must be logged on to the computer that either maps to that user or is included in the specified group.

See Also

Configuring MTS Roles for Declarative Security, Packages Installed Folder

Activation Tab (Package)

The **Activation** tab is used to determine how components are activated in your package.

You can select a package to run in process of the client that called it (as a Library package) or in a dedicated local process (as a Server package).

Library Package

Select this option to run the package as a Library package. A Library package runs in the process of the client that creates it. This option is only available for clients on the computer on which the package is being installed and configured. Note that Library packages offer no component tracking, role checking, or process isolation.

Server Package

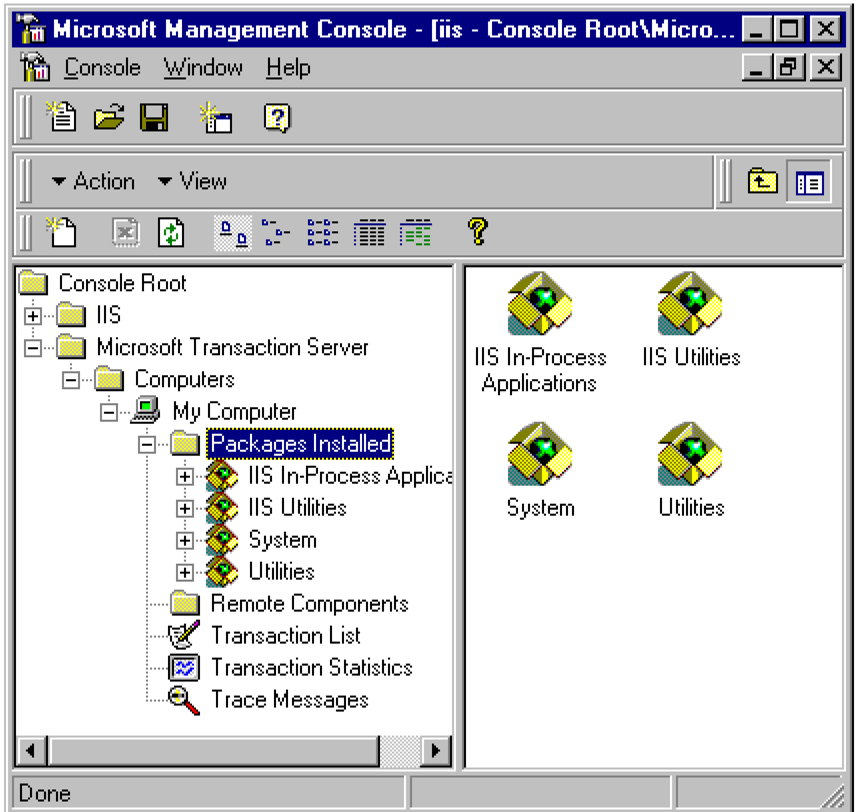
Select this option to run this package as a Server package. A server package runs in its own process on the local computer. Server packages support role-based security, resource sharing, process isolation, and process management (such as package tracking).

See Also

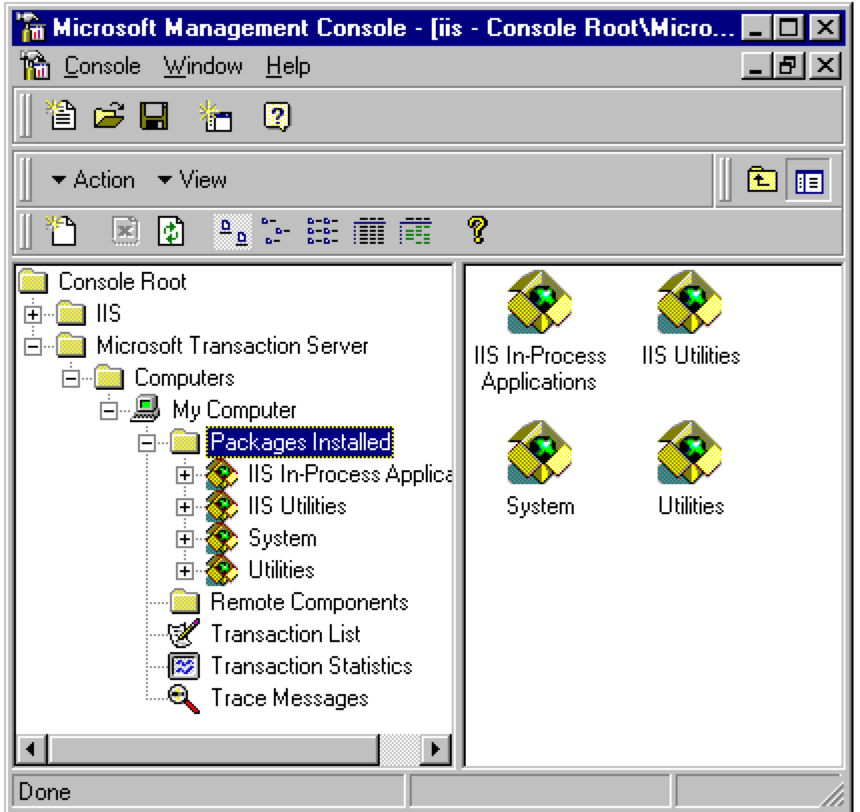
Packages Installed Folder, Package Properties

Component Properties

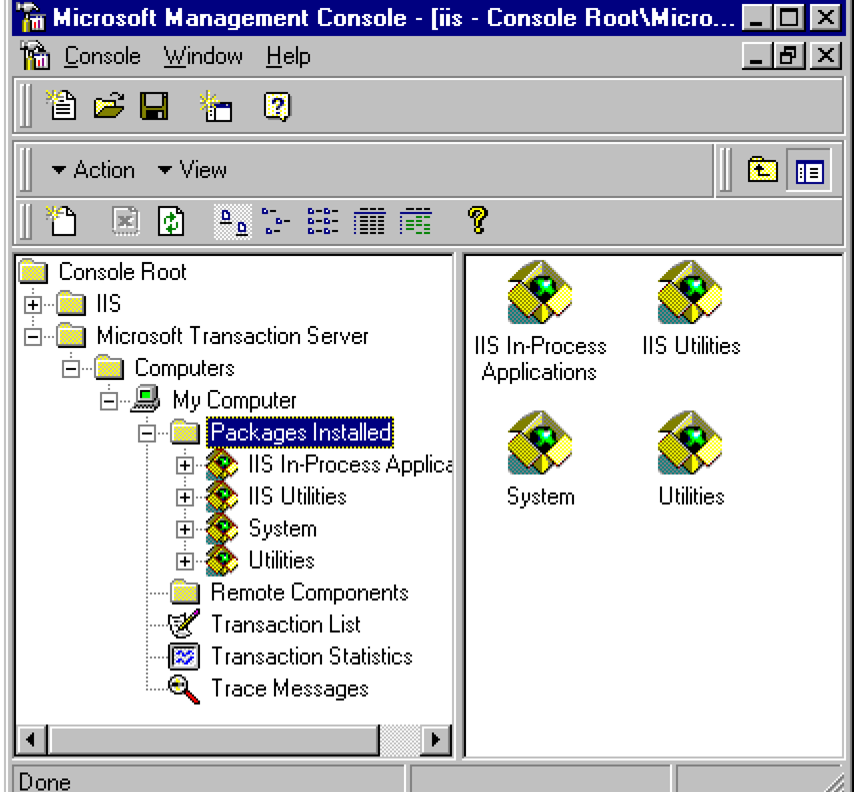
Component properties control transaction support and security settings. You can also use the **Properties** tab to view the component's identifying properties, such as its Name, programmatic identifier (ProgID), and class identification (CLSID).



General



Transaction



Security

See Also

Components Folder, Enabling MTS Package Security

General Tab (Component)

The **General** tab displays general information about the selected component.

Component ProgID

Displays the programmatic identifier (ProgID).

Description

Displays a description of the component. You can type a description to identify and manage the component.

DLL

Displays the path of the DLL containing the class and interface definitions for the component.

CLSID

Displays the unique class identifier (CLSID) of the selected component. You can use a CLSID in code to identify and access a component.

Package

Displays the name of the package where the selected component is installed.

See Also

Components Folder, Component Properties

Security Tab (Component)

The **Security** tab is used to configure security for the selected component. See the Enabling MTS Package Security topic for learn about declarative security.

Enable authorization checking

Select this box to check the security credentials of any client that calls the component.

See Also

Components Folder, MTS Component Properties

Transaction Tab (Component)

The **Transaction** tab determines how a component supports transactions.

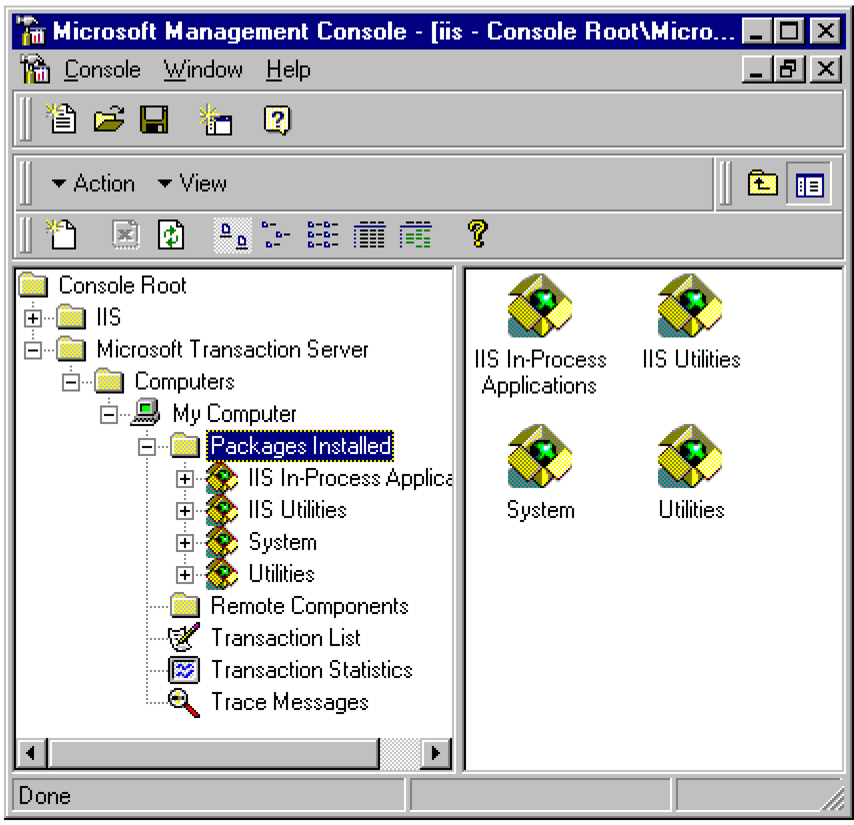
- **Requires a transaction** — This indicates that the component's objects must execute within the scope of a transaction. When a new object is created, its object context inherits the transaction from the context of the client. If the client does not have a transaction, MTS automatically creates a new transaction for the object.
- **Requires a new transaction** — This 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 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 indicates that the component's objects should 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.

See Also

Components Folder, MTS Component Properties, Managing MTS Transactions

Remote Component Properties

Remote component properties are used to display information about the components that have been added to the Remote Components folder. You cannot configure remote component properties other than to provide a description of the component in the **General** tab.



General Tab

See Also

Remote Components Folder, Distributing MTS Packages

General Tab (Remote Component)

The **General** tab displays identification information about the selected remote component.

Name

Displays the name of the remote component.

Description

Displays a description of the remote component. You can type a description to help you identify and manage the remote component.

CLSID

The class ID for the component.

Runs On

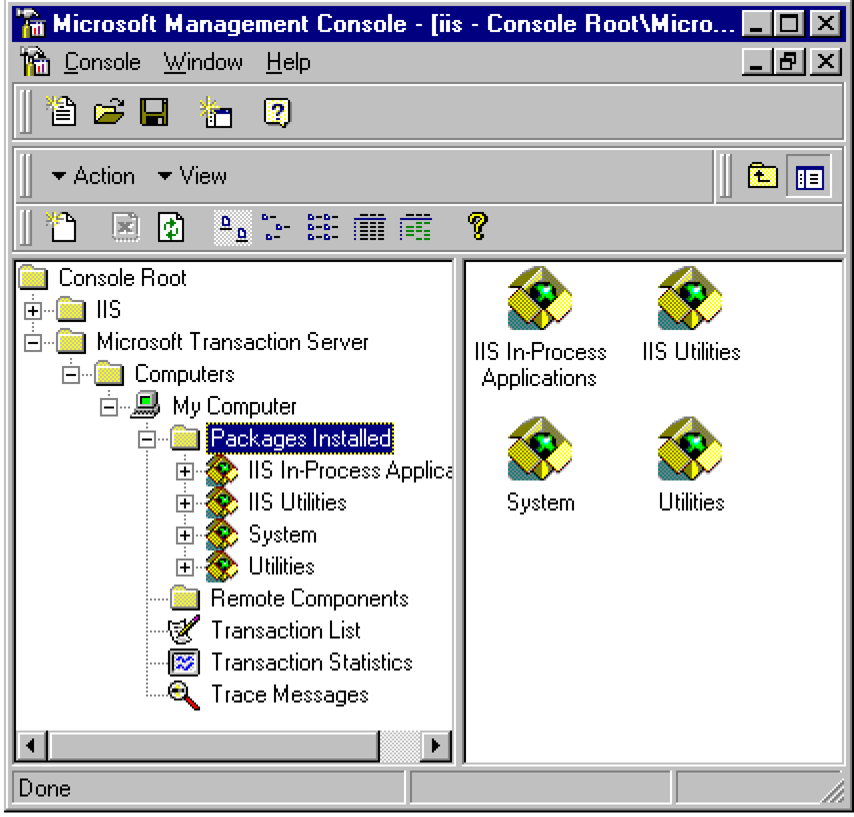
Identifies the name of the computer the component was installed from.

See Also

Remote Components Folder

Role Properties

Role properties are used to view the name, description, and Role ID for a particular role.



General

See Also

Roles Folder, Enabling MTS Package Security, Mapping MTS Roles to Users and Groups

General Tab (Role)

The **General** tab displays general information about the selected role.

Name

Displays the name of the role.

Description

Displays a description of the role. You can type a description to help you identify and manage the role.

Role ID

Displays the role identification number that MTS generates when you add a role. You can use this number to distinguish between roles with similar names that you have identified for different packages.

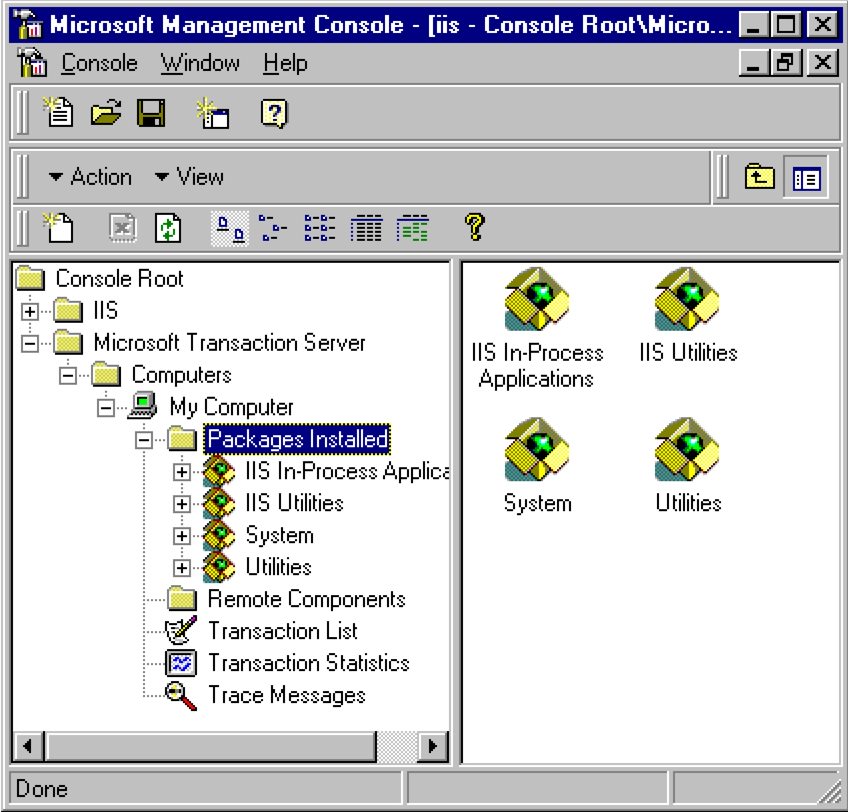
See the Roles Folder topic for an overview of the objects in the Roles folder and the functions performed at the role level.

See Also

Roles Folder, MTS Role Properties, Enabling MTS Package Security, Mapping MTS Roles to Users and Groups

Role Member Properties

Role member properties are used to display information about the roles that have been added to a component or interface.



General

See Also

Roles Membership Folder, Enabling MTS Package Security, Mapping MTS Roles to Users and Groups

General Tab (Role Member)

The **General** tab displays general information about the selected role.

Name

Displays the name of the role.

Description

Displays a description of the role. You can type a description to help you identify and manage the role.

Role ID

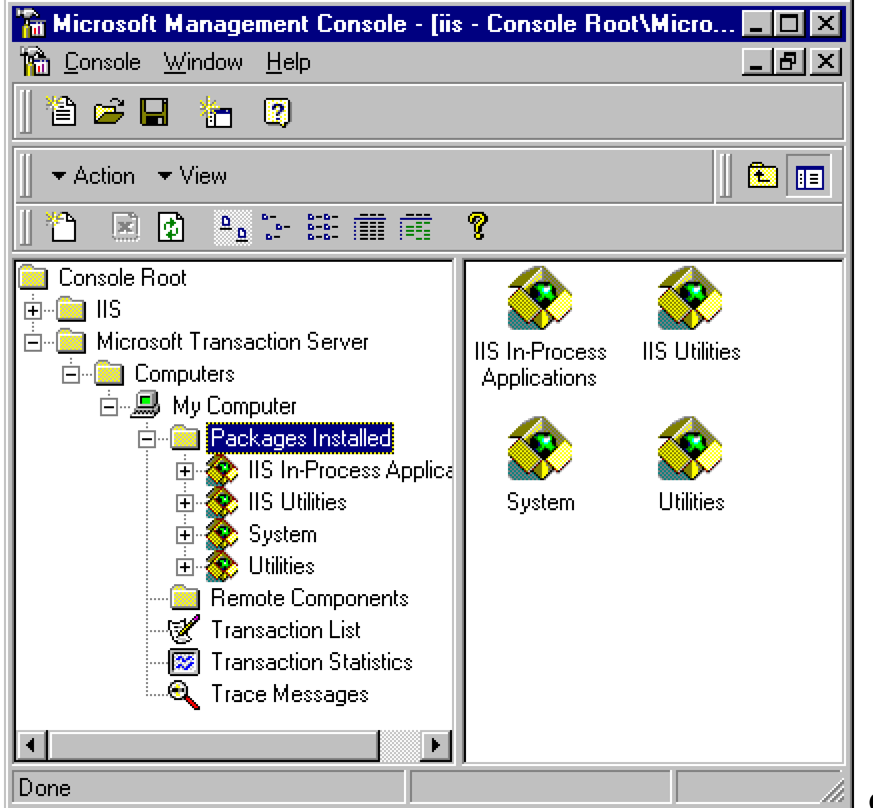
Displays the role identification number that MTS generates when you add a role. You can use this number to distinguish between roles with similar names that you have identified for different packages.

See Also

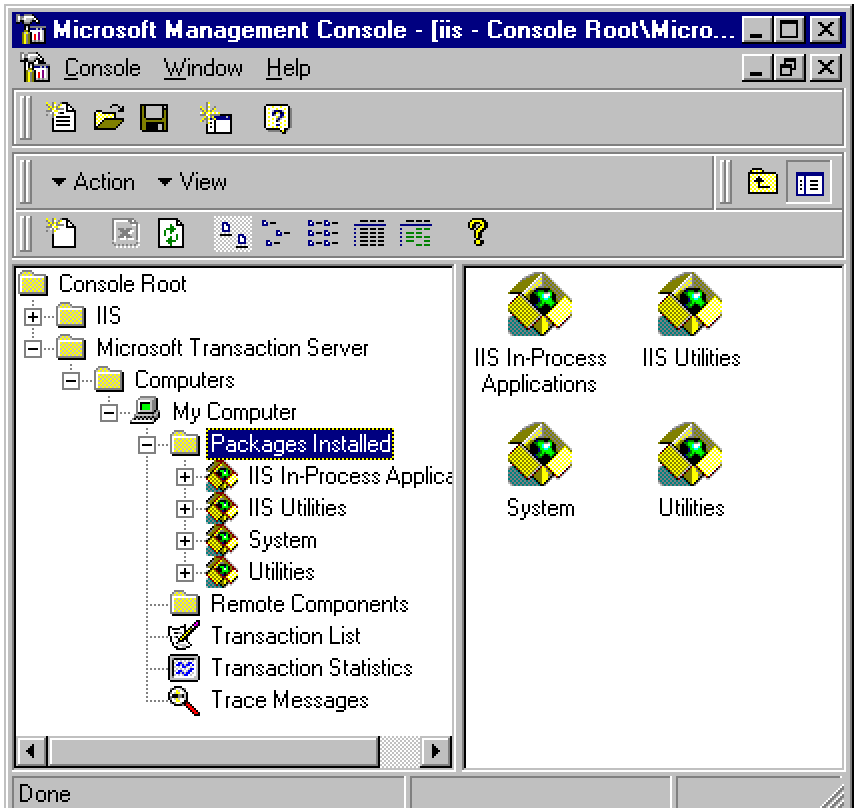
Roles Membership Folder, Role Properties, Enabling MTS Package Security, Mapping MTS Roles to Users and Groups

Interface Properties

Interface properties are used to display information about an interface exposed by a component.



General



Proxy

See Also

Interfaces Folder

General Tab (Interface)

The **General** tab displays identification information about the selected interface.

Name

Displays the name of the interface.

Description

Displays a description of the interface. You can type a description to help you identify and manage the interface.

IID

Displays the interface identifier.

See Also

Interfaces Folder, Interface Properties

Proxy Tab (Interface)

Displays identification information about the selected proxy/stub.

Proxy/Stub

Displays the CLSID and file name of the proxy/stub DLL.

Type Library

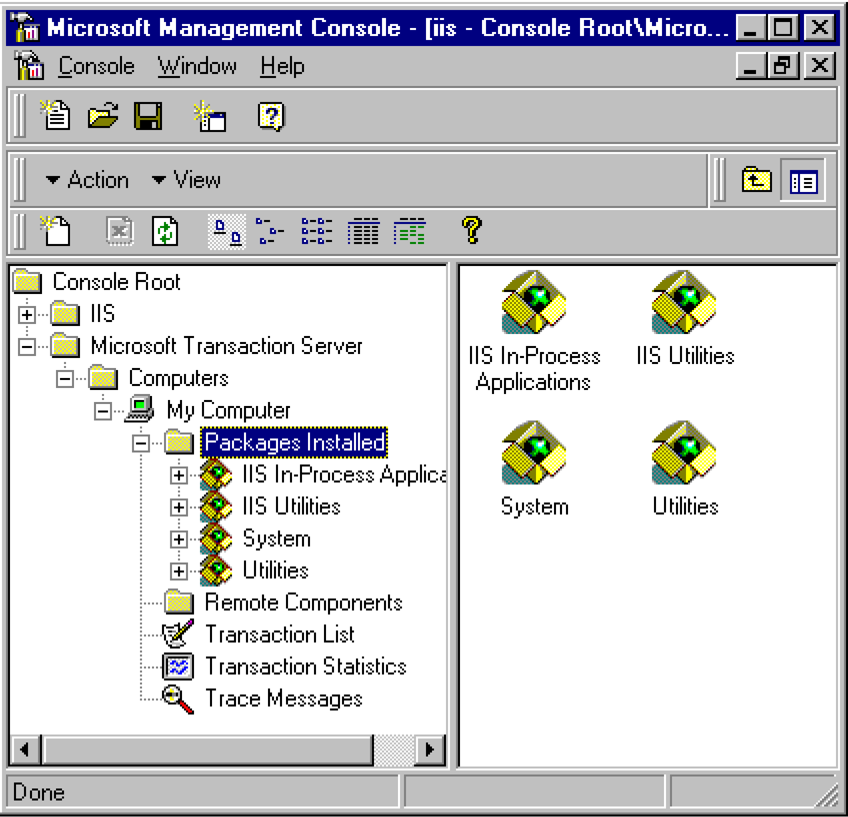
Displays the Library ID and file location for the type library.

See Also

Interfaces Folder, Interface Properties

Method Properties

Method properties are used to display information about the methods exposed by an interface.



General

See Also

Methods Folder

General Tab (Method)

The **General** tab displays identification information about the selected method.

Name

Displays the name of the method.

Description

Displays a field in which you can type a description of the method.

See Also

Methods Folder, Method Properties

Creating MTS Packages

Creating packages is the final step in the MTS application development process. Package design decisions dictate the organization and properties of components. Although creating MTS packages does not require programming knowledge, you should be thoroughly familiar with the design and implementation specifications of the application.

It is highly recommended that you review this section of the *MTS Administrator's Guide* in conjunction with the *MTS Programmer's Guide* so that you understand the design-time implications for packaging components using the Microsoft Transaction Server Explorer.

Packaging components enacts development decisions that include resource pooling, activation settings, and support for transactions. For example, when you create a package, you should try to group components that share resources in the same package. Consider the type of resources that components are sharing in your package, and group components that share "expensive" resources, like connections to a specific database.

Packaging components to take advantage of resource pooling results in more efficient MTS applications. If you review the *MTS Programmer's Guide* in conjunction with the *MTS Administrative Guide*, you can learn more about the development concerns that motivate creating and populating packages using the Explorer. See the Installing MTS Development Samples and Documentation topic for instructions on obtaining the *MTS Programmer's Guide*.

This section discusses the following topics:

Creating an Empty MTS package

Adding a Component to an MTS Package

Importing a Component into an MTS Package

Removing a Component from an MTS Package

Building an MTS Package for Export

Setting MTS Package Properties

Setting MTS Activation Properties

Setting MTS Transaction Properties

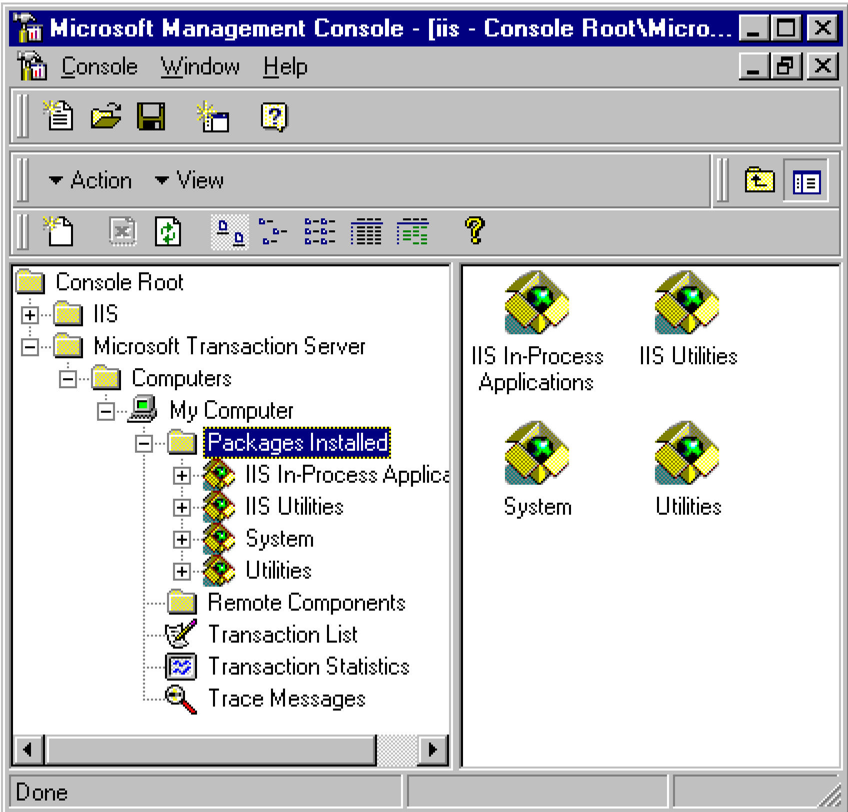
Setting MTS Authentication Levels

Locking Your MTS Package

Creating an Empty MTS Package

The first step in using the MTS Explorer is to create a package. A package, which can easily be created in the MTS Explorer, is a collection of components that run in the same process. You can either create an empty package and then add components, or you can install a pre-built package. Add packages by using the Package wizard or by dragging a package file (.pak) from the Windows NT Explorer and dropping it into the right pane of the MTS Explorer.

Packages define the boundaries for a server process running on a server computer. For example, if you group a sales component and a purchasing component in two different packages, these two components will run in separate processes with process isolation. Therefore, if one of the server processes terminates unexpectedly (such as an application fatal error), the other package can continue to execute in its separate process.



To create an empty package

1. In the left pane of MTS Explorer, select the computer for which you want to create a package.
2. Open the Packages Installed folder for that computer.
3. On the **Action** menu, click **New**. You can also select the Package Installed folder and either right-click and select **New** and then **Package** from the right-click menu, or select the **Create a new object** button on the MTS toolbar.
4. Use the Package wizard to install either a pre-built package or create an empty package. If you create an empty package, you must add components and roles before it will be functional.
5. Click the **Create an empty package** button.
6. Type a name for the new package, and click **Next**.
7. Specify the package identity in the **Set Package Identity** dialog box, and then click the **Finish** button.

The default selection for package identity is **Interactive User**. The interactive user is the user that logged on to the server computer on which the package is running. You can select a different user by

selecting the **This user** option and entering a specific Windows NT user or group.

See Also

Adding a Component to an MTS Package, Importing a Component into an MTS Package, Building an MTS Package for Export, Enabling MTS Package Security

Adding a Component to an MTS Package

An MTS component is a reusable piece of code and data that is built to the Component Object Model (COM) specification. Components enact business logic in an application.

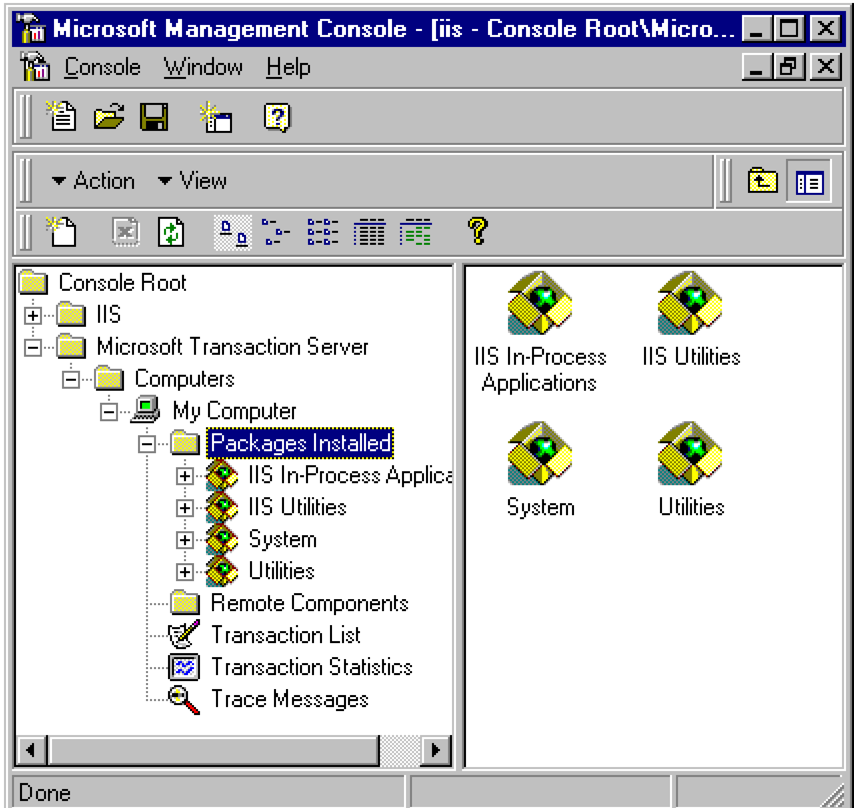
You can add a component to a package by:

- using the component wizard
- moving a component from an existing package

To add components to the Components folder of a package, you can either use the component wizard in MTS Explorer or you can drag dynamic-link libraries (DLL) that contain the components you want from Windows NT Explorer and drop them in the package . If you use the Component wizard, you can either install a new component, which adds the component to the system registry, or import components that have already been registered. Components can be added to empty packages or existing packages.

To move a component from an existing package, simply drag and drop the component from the existing package to the new component.

Note that a single MTS application can contain components that can be installed in multiple packages. You can place different components that are housed in the same DLL into completely separate packages.



To add a component to a package

1. In the left pane of MTS Explorer, select the computer on which you want to install the component.
2. Open the Packages Installed folder, and select the package in which you want to install the component.
3. Open the Components folder.
4. On the **Action** menu, click **New**. You can also select the Components folder, right-click, and select

New and then **Component** from the right-click menu, or select the **Create a new object** button on the MTS toolbar.

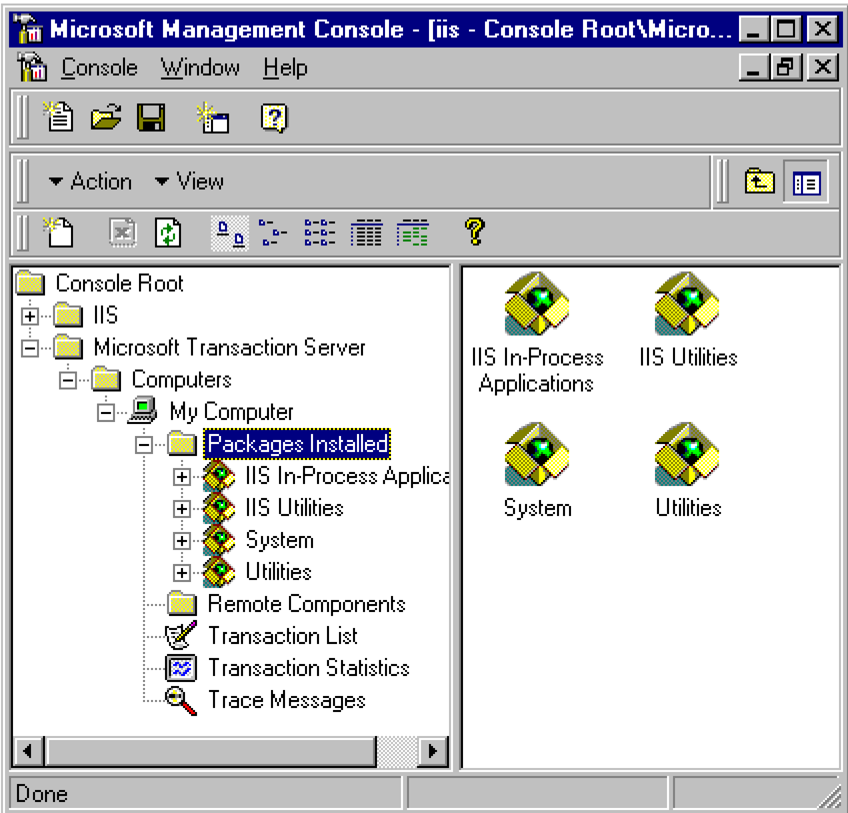
5. Click the **Install new component(s)** button.
6. In the dialog box that appears, click **Add Files** to select the files you want to install. You should select the DLL that contains the component you want to install. If the component has an external type library or proxy/stub DLL, also add those files. Make sure that in your Windows NT Explorer, the **Hidden files** option is set to **Show all files**. If this option is set to hide files with the .dll file name extension, you will not see the DLLs that contain your component in the Component wizard **Add Files** dialog box. You will have to restart the MTS Explorer if you change this setting.
7. In the dialog box that appears, select the file or files you want to add, and click Open. You can display all available files, just DLLs, or just type libraries by clicking the appropriate option in the **Files of type** box.
8. After you add the files, the Install Components dialog box displays the files you have added and their associated components. If you select the Details check box, you will see more information about file contents and the components that were found. Microsoft Transaction Server components must have a type library. If MTS cannot find your component's type library, your component will not appear in the list. You can also remove a file from the **Files to install** list by selecting it and clicking Remove Files.
9. Click the **Finish** button to install the component. It is important to note that installing a component allows you to view the interfaces and methods on that component. When you import a component, the imported component's interfaces and methods are not visible in MTS Explorer.

See Also

Creating an Empty MTS Package, Importing a Component into an MTS Package, Removing a Component from an MTS Package, Building an MTS Package for Export

Importing an MTS Component into a Package

You can use the Microsoft Transaction Server Explorer to import into packages specific components that have already been registered on your computer as COM (Component Object Model) components. Importing a component does not install the interface or method information required to set interface properties or to configure access to the component from a remote client. If possible, install rather than import components.



To import a component into a package

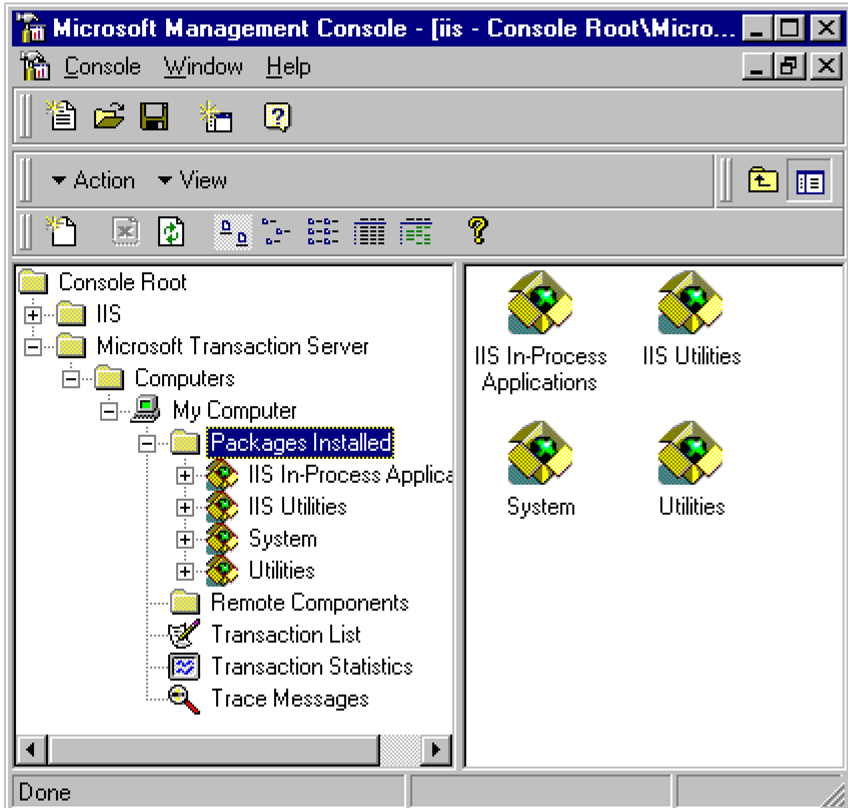
1 In the left pane of MTS Explorer, select the computer on which you want to import the component.
2 Open the Packages Installed folder, and select the package into which you want to import the component.
3 Open the Components folder.
4 On the **Action** menu, click **New**. You can also select the Components folder and either right-click and select **New** and then **Component** from the right-click menu, or select the **Create a new object** button on the MTS toolbar.
5 Click the **Import component(s) that are already registered** button.
6 Select the components you want to import.
7 Click **Finish**.

See Also

Creating an Empty MTS package, Adding a Component to an MTS Package

Removing an MTS Component from a Package

You can use Microsoft Transaction Server Explorer to remove components from a package. The impact of deleting a component depends on how the component was added to the MTS run-time environment. If you installed the component, deleting the component completely removes registry information from both the MTS run-time environment and your computer. If you imported the component, the component will be removed from the MTS run-time environment, but will remain registered on the deployment computer as a COM (Component Object Model) component. You must then manually remove the COM component registry entries and component files from your deployment computer.



To remove a component from a package

1 In the right pane of the Explorer, select the package that contains the component you want to remove.

2 Open the Components folder.

3 Select the component you want to remove.

4 On the **Action** menu, click **Delete**. You can also select the component and either right-click and select **Delete** from the right-click menu, or select the **Delete** button on the MTS toolbar.

5 Click **Yes** in the dialog box that appears.

See Also

Creating an Empty MTS package

Building an MTS Package for Export

To export packages, your components must be properly configured so that MTS can automate package registration on a server or client computer. When building a package, therefore, you should consider how the package might be distributed.

You can use the package export option in the Microsoft Transaction Server Explorer to export your package to another server computer running MTS. You can also generate application executables for remote client computers running Windows NT or Windows 95 (with DCOM support) to access your server application. Consider the requirements for package export while you are creating and configuring packages.

By default, application executables configure client machines to access the remote MTS server on which the executable was generated. You can modify the location of the server application by configuring the **Option** tab of the Computer property sheets. Before you generate the executable, select **My Computer** in the MTS Explorer, right-click, and choose **Properties** from the right-click menu. Click the **Options** tab and enter the machine name of the server that you want the client computer to access in **the Remote server name box**. Note that the machine name that you enter must be the MTS server running the package. Then click **OK**. When you generate the client executable, the executable will configure that client to access the server that you specified in the **Remote server name** box. This allows you to statically load balance your application by having multiple clients point to more than one machine running the same package.

Requirements for Package Export

Package developers, or advanced system or Web administrators who deploy packages must observe the following requirements while building and deploying MTS packages:

- Remove the descriptions of standard COM interfaces from the client-only application type libraries. For example, the package developer may have defined an interface such as **IObjectSaftey** in a type library in order to use that interface with Visual Basic. Removing descriptions of the interface before exporting will prevent the interface from being improperly registered and unregistered on client machines. Failure to remove standard COM interface descriptions from the client-only type library could lead to the failure of any other application using those standard interfaces.
- If any of the globally unique identifiers (GUIDs) that are in the server package (including class, interface, or type-library identifiers) and are used by clients change, you will need to re-export the package if you want to generate an updated client install executable. Clients of your application will not be able to access the server application until they run the new client install executable. Note that some development tools (such as Microsoft® Visual Basic™) may change these GUIDs without notifying the developer.

For more information about using the MTS Explorer to distribute MTS packages, see the Distributing MTS Packages section.

See Also

Creating an Empty MTS package, Adding a Component to an MTS Package, Importing a Component into an MTS Package

Setting MTS Package Properties

Package properties determine package configuration. To access these property sheets, right-click the package in the Microsoft Transaction Server Explorer, or select the package and choose the **Properties** option from the **Action** menu.

There are five package property sheets:

- **General**

  Displays the name and description the computer.
- **Security**

  Enables authorization checking. The default security setting enables authorization checking. See the Enabling MTS Package Security topic for more information.
- **Advanced**

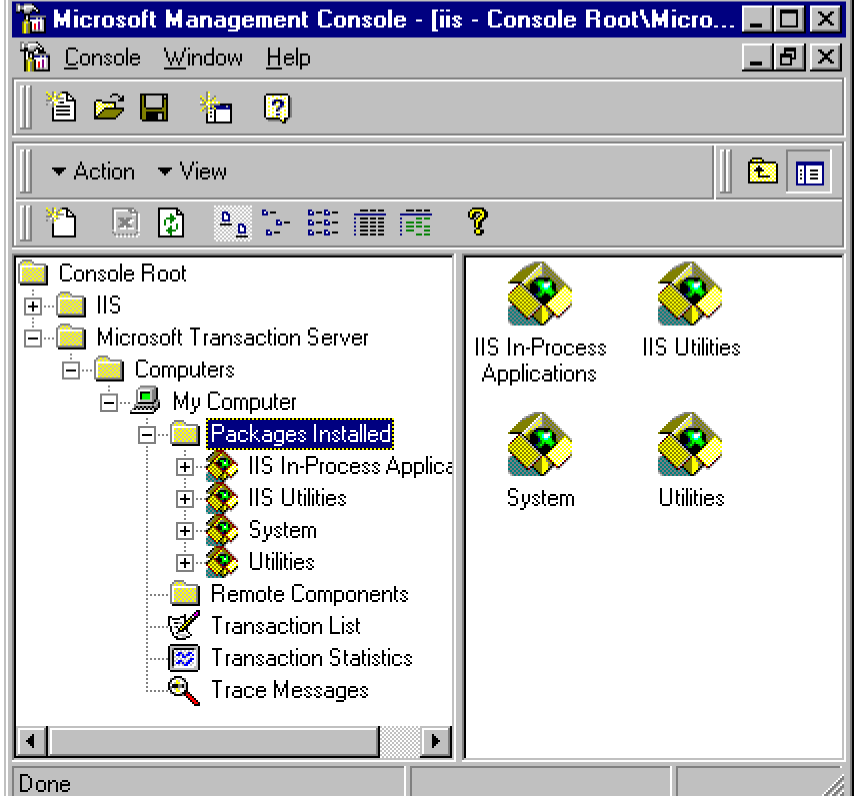
  Determines whether the server process associated with a package always runs, or whether it shuts down after a certain period of time.
- **Identity**

  Used to set which user is allowed to access a package. The default value is Interactive User, which is the user that is currently logged on to the Windows NT server account. If you want to select another user, you can select the **This user** option and specify an account name and password.
- **Activation**

  Used to set the activation level for the package and its components. You can either set the activation level to **Library Package** so components are activated in the creator's process, or to **Server Package** so that the package runs in a dedicated server process.

  See the Setting MTS Activation Properties topic for more information.

To modify package or component properties, you must use property sheets.

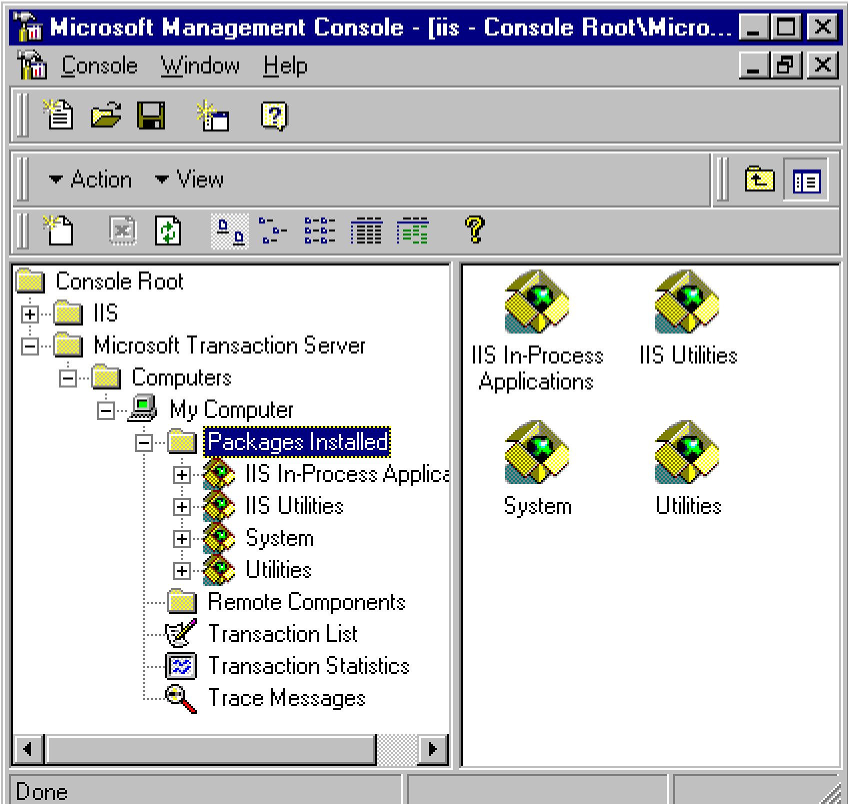


To access property sheets:

1 Right-click the package or component that you would like to configure and choose **Properties**. You can also select the item, open the **Action** Menu, and select **Properties**.
2 Select the tab of the sheet that you will use.
3 Update the property setting.
4 Click **OK** to save the setting and return to the Explorer.

Refreshing Component Settings

It is important to refresh the MTS settings for components each time you recompile your project. Refreshing component settings prevents your component registry settings from being rewritten.



To refresh your component settings

1 In the left pane of the MTS Explorer, select the computer that contains the components you would like to refresh.
2 On the **Action** menu, click **Refresh All Components**. This updates Transaction Server with any changes to the system registry, component CLSIDs, or interface identifiers (IIDs). You can also refresh components by selecting the computer in the left pane of the Explorer and clicking the **Refresh** button on the MTS toolbar.

   The **Refresh All Components** command works if any item below a computer in the MTS Explorer hierarchy has been selected. The command will apply to the selected computer.

See Also

Setting MTS Activation Properties, Setting MTS Transaction Properties, Setting MTS Authentication Levels, Locking Your MTS Package

Setting MTS Activation Properties

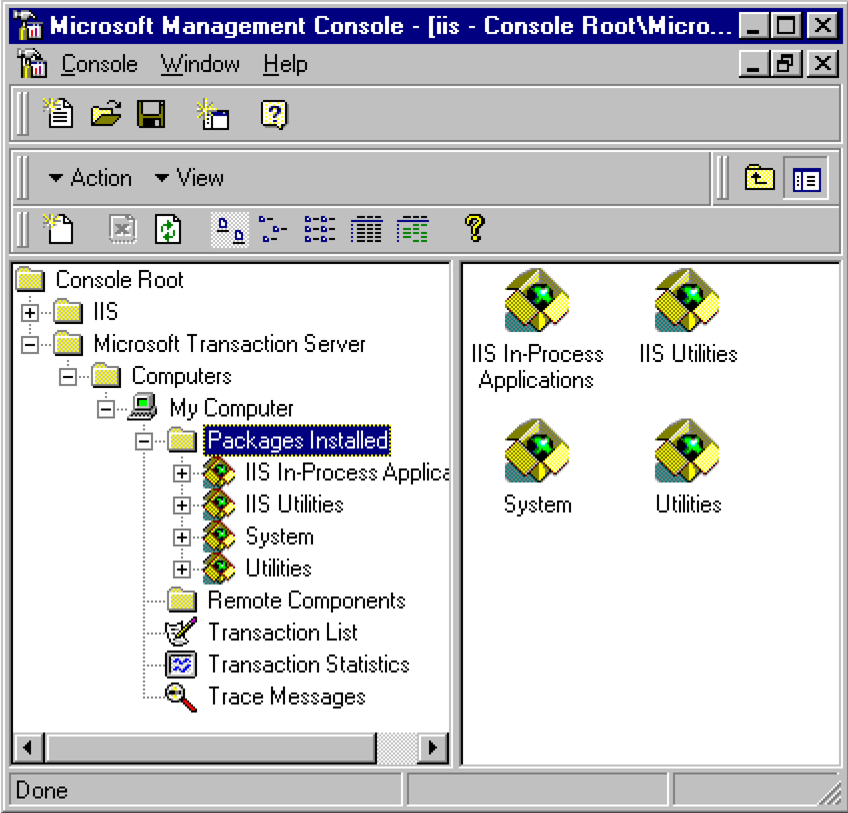
Activation properties should be determined at package design time. You can select a package to run in process of the client that called it (as a library package) or in a dedicated server process (as a server package).

Library Package

Select this option to run the package as a library package. A library package runs in the process of the client that creates it. This option is only available for clients on the computer on which the package is being installed and configured. Note that library packages offer no component tracking, role checking, or process isolation.

Server Package

Select this option to run this package as a server package. A server package runs in its own process on the local computer. Server packages support role-based security, resource sharing, process isolation, and process management (such as package tracking).



To set the package activation property

1. Select the package you want to configure.
2. On the **Action** menu, click **Properties**, and select the **Activation** tab. You can also access the property sheets by selecting the item and either right-clicking and choosing **Properties**, or clicking the **Properties** button on the MTS toolbar.
3. Click the **Activation** tab and specify the appropriate activation property.
4. Click **OK** to return to the Explorer.

See Also

Setting MTS Package Properties, Setting MTS Transaction Properties, Setting MTS Authentication Levels, Locking Your MTS Package

Setting MTS Transaction Properties

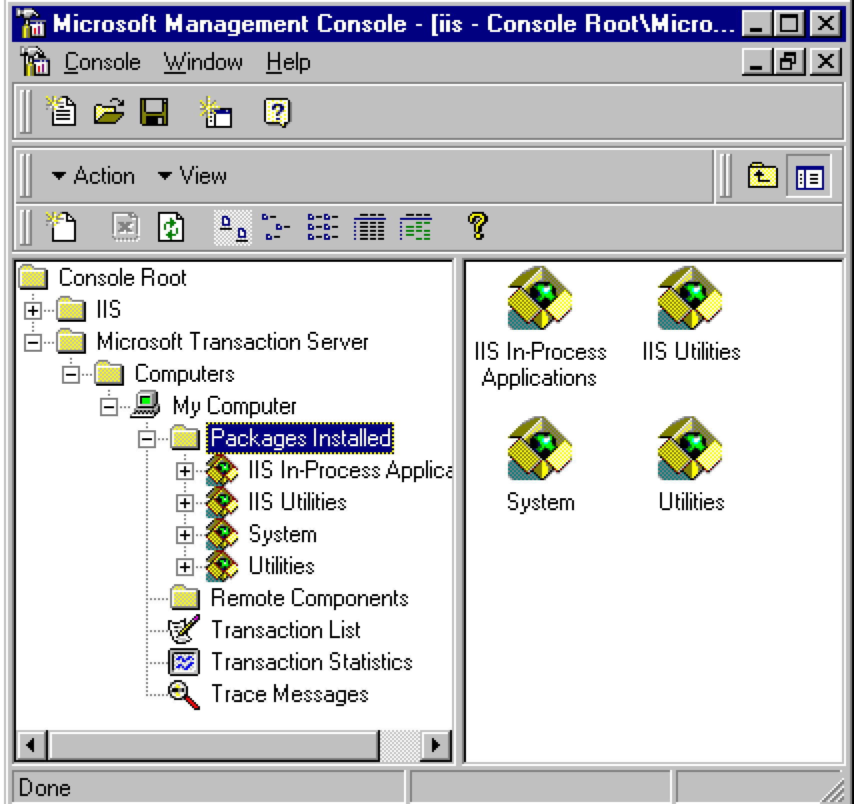
You should determine if you want your application to support transaction at design time, and then use the Microsoft Transaction Server Explorer during deployment to set the component transaction property. The **Transaction** tab on the property sheet determines how a component supports transactions. You can select one of the following settings for component transaction properties:

- **Requires a transaction** This 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 doesn't have a transaction, MTS automatically creates a new transaction for the object.
- **Requires a new transaction** This indicates that the component's objects must execute within their own transactions. When a new object is created, Transaction Server automatically creates a new transaction for the object, regardless of whether its client has a transaction.
- **Supports transactions** This 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 doesn't have a transaction, the new context is also created without one.
- **Does not support transactions** This indicates that the component's objects shouldn't 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.

Whenever an instance of a component is created, MTS checks the component's transaction attribute to determine whether the instance should run in a transaction.

Not all components are designed to support transaction processing. If your component is not designed to use transaction processing, make sure that the transaction attribute on the component's **Transaction** tab is set to **Does not support transactions**.

Note that you cannot modify transaction attributes if components in the package have been *locked*. For more details, see Locking Your Package.



To set component transaction properties

1. Select the component you want to configure.
2. On the **Action** menu, click **Properties**, and select the **Transaction** tab. You can also access the property sheets by selecting the item and either right-clicking and choosing **Properties**, or clicking the **Properties** button on the MTS toolbar.
3. Click the appropriate transaction attribute box.
4. Click **OK**.

For an overview of distributed transactions and instructions on how to monitor and manage transactions, see the Managing MTS Transactions.

See Also

Setting MTS Package Properties, Setting MTS Activation Properties, Setting MTS Authentication Levels, Locking Your MTS Package

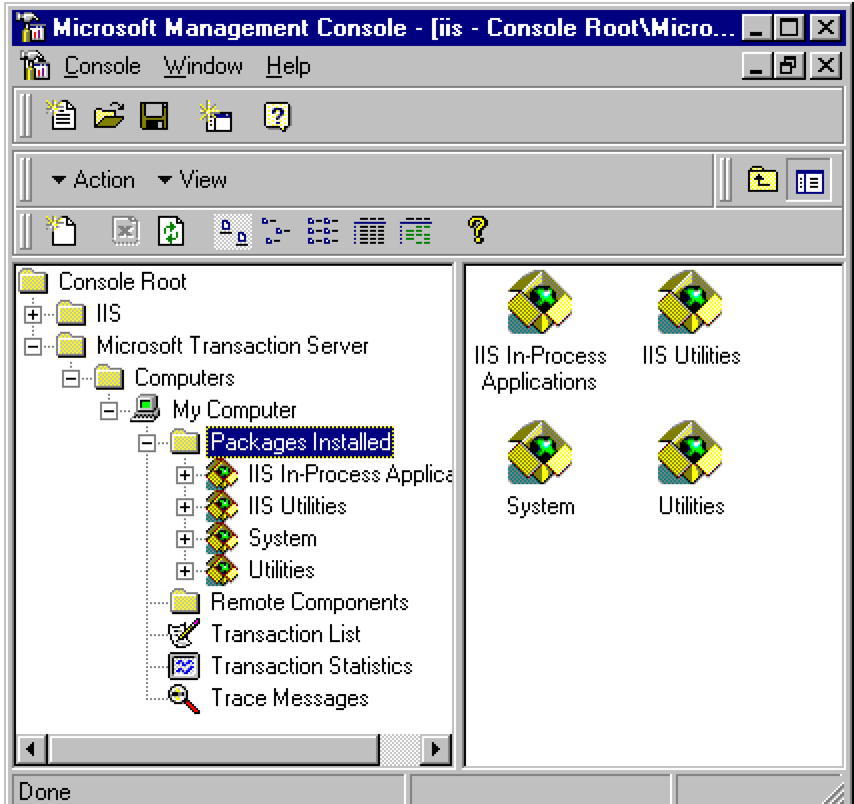
Setting MTS Authentication Levels

An application's authentication level indicates the level of security used to authenticate client requests. If authentication is not required for clients of an application, select the **Anonymous** option on the **Security** tab of the Package property sheet. If you want to require authentication of clients, use the **Impersonate** setting. The server process will set the authentication level to that specified level and the impersonation level to impersonate so it is not affected by distributed COM (DCOM).

Unless you have a thorough understanding of DCOM authentication levels, it is recommended that you leave your package authentication setting at the MTS default setting, which is Packet level.

The following table describes the different DCOM authentication settings:

Level	Description
None	No security checking occurs on communication between this package and another package or a client application.
Connect	Security checking occurs only for the initial connection.
Call	Security checking occurs on every call for the duration of the connection.
Packet	The sender's identity is encrypted to ensure the .
Packet Integrity	The sender's identity and signature are encrypted to ensure that packets haven't been changed during transit.
Packet Privacy	The entire packet, including the data, and the sender's identity and signature, are encrypted for maximum security.



To set authentication levels for a computer

1. In the MTS Explorer, select the package you want to configure.
2. In the **Action** menu, click **Properties**, and select the **Security** tab. You can also access the property sheets by selecting the package and either right-clicking and choosing **Properties**, or clicking the **Properties** button on the MTS toolbar.
3. Select the level of authentication you want to configure for this package under Authentication level for calls.
4. Click **OK**.

See Also

Setting MTS Package Properties, Setting MTS Activation Properties, Setting MTS Transaction Properties, Locking Your MTS Package

Locking Your MTS Package

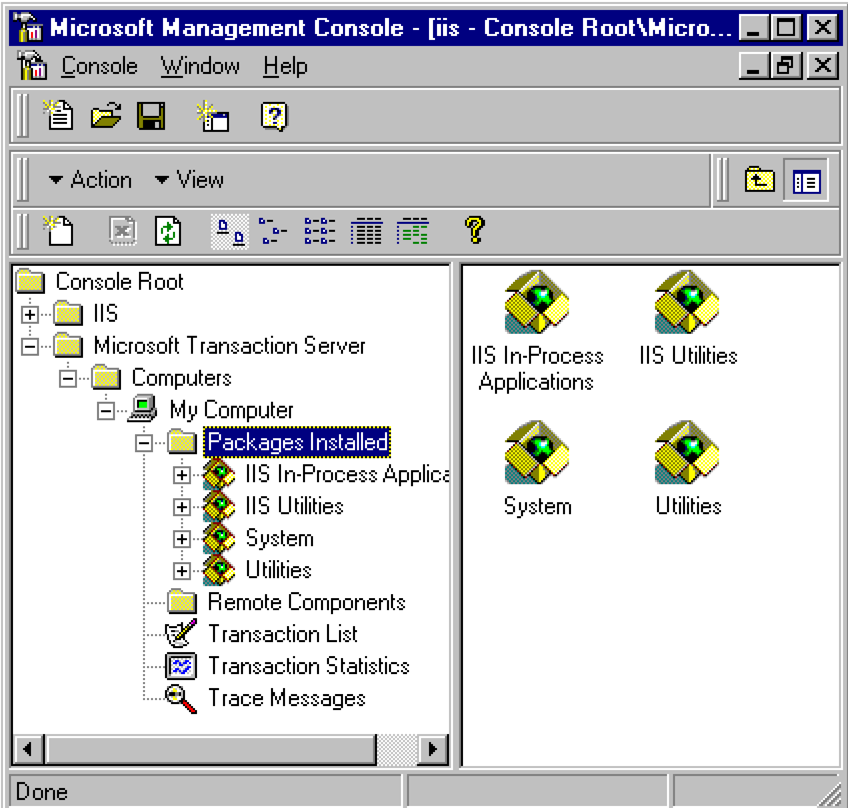
After you have developed, installed, or deployed your application, consider locking your package so component configuration cannot be modified. You can lock your package before exporting a server application to another MTS computer that is maintained by a system or Web administrator, or before you distribute a server application to customers.

You can lock your package against:

- Changes

  This setting ensures that administrators cannot modify package configuration without disabling the lock.
- Deletions

  This setting ensures that administrators cannot delete a package without disabling the lock.



To lock your package

1. Once you have finished configuring your package, select the **Advanced** tab in the package property sheets.
2. In the Permissions section of the **Advanced** property sheet, select either the **Disable deletion** option (which prohibits users from deleting the package) or the **Disable changes** option (which blocks users from making any changes to the package properties.

You can unlock a package by clearing the **Advanced** property sheet check-boxes.

See Also

Setting MTS Package Properties, Setting MTS Activation Properties, Setting MTS Transaction Properties, Setting MTS Authentication Levels

Distributing MTS Packages

You can use the Microsoft Transaction Server Explorer to distribute packages to clients running MTS as well as to clients that are not running MTS. If both the client and server computer are running MTS, you can use the MTS Explorer to create application executables, or push and pull components using the Remote Components folder. If the client computer is not running MTS, use the MTS Explorer to generate application executables that automatically install and configure clients to access remote MTS server applications over distributed COM (DCOM).

This section discusses the following topics:

Working with Remote MTS Computers

Exporting MTS Packages

Generating MTS Executables

See Also

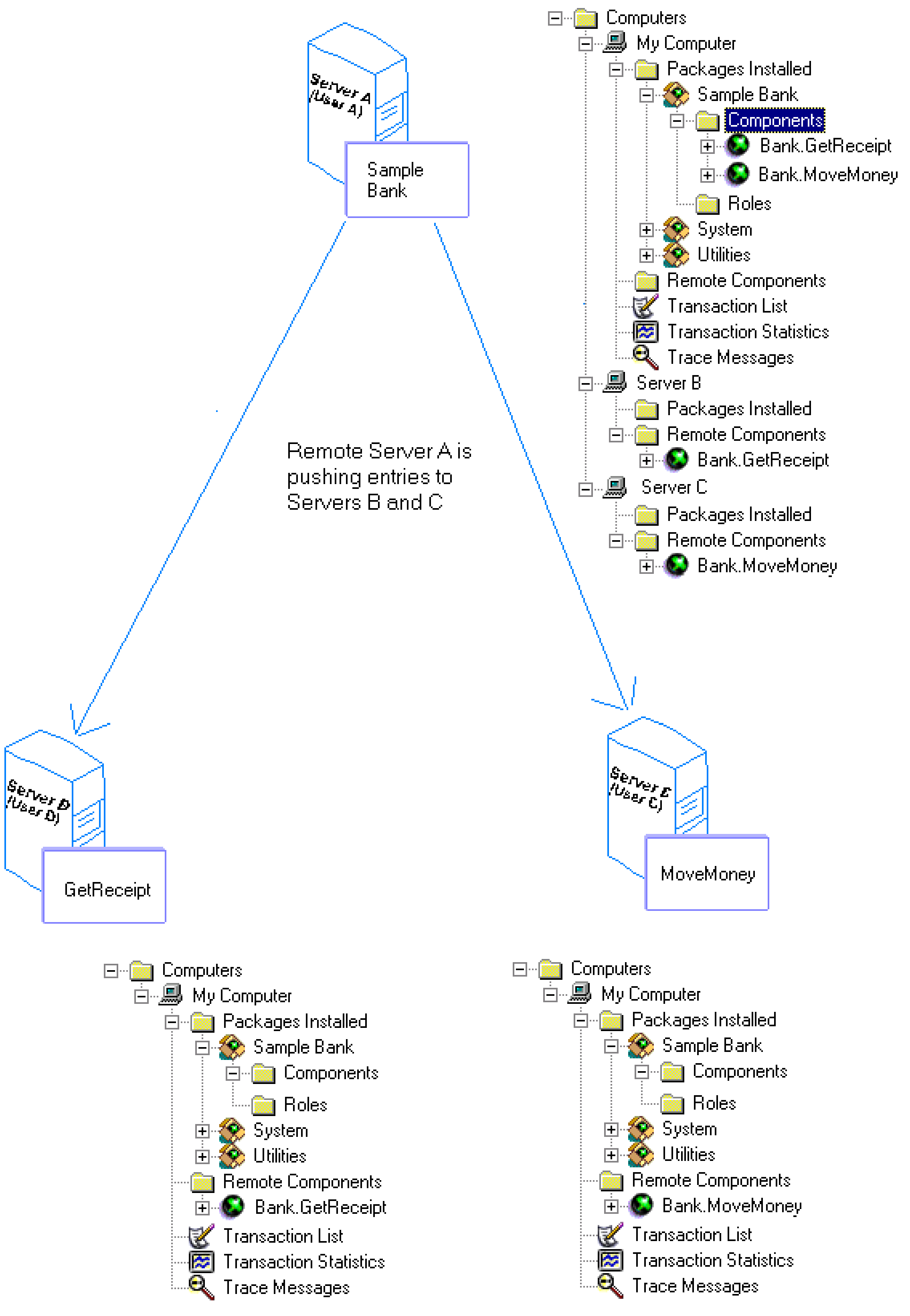
MTS Remote Components Folder

Working with Remote MTS Computers

If both the server and client computer are running MTS, you can distribute a package by "pulling" and "pushing" components between one or more computers. You can also generate an application executable using the application executable utility in the Microsoft Transaction Server Explorer. See the Generating MTS Executables topic for more information on using the MTS Explorer to generate application executables.

Pushing components means creating remote component entries on remote computers. Once the remote component entries are created, you have to add those component entries to your Remote Components folder on your local machine (pull the components).

The following diagram illustrates how pushing and pulling components configure a remote computer running MTS.


Server A (User A)
Sample Bank
Computers
My Computer
Packages Installed
Sample Bank
Components
Bank.GetReceipt
Bank.MoveMoney
Roles
System
Utilities
Remote Components
Transaction List
Transaction Statistics
Trace Messages
Server B
Packages Installed
Remote Components
Bank.GetReceipt
Server C
Packages Installed
Remote Components
Bank.MoveMoney
Remote Server A is pushing entries to Servers B and C
Server B (User B)
GetReceipt
Server C (User C)
MoveMoney
Computers
My Computer
Packages Installed
Sample Bank
Components
Roles
System
Utilities
Remote Components
Bank.GetReceipt
Transaction List
Transaction Statistics
Trace Messages
Computers
My Computer
Packages Installed
Sample Bank
Components
Roles
System
Utilities
Remote Components
Bank.MoveMoney
Transaction List
Transaction Statistics
Trace Messages

In the diagram, MTS is installed on all three computers. GetReceipt and MoveMoney are both installed in a package on your computer (Server A). When you add the GetReceipt and MoveMoney components to the Remote Components folders of Server B and Server C, two things happen. First, the appropriate DLL files are copied across the network to Server B and Server C. These files will be copied to a subdirectory with the same name as the Package on Server A. This subdirectory will be created under the \<*Microsoft Transaction Server Install directory*>\Remote directory (for example, C:\Program Files\MTx\Remote\Sample Bank). Second, the Server B and Server C system registries are updated with information from the system registry on Server A.

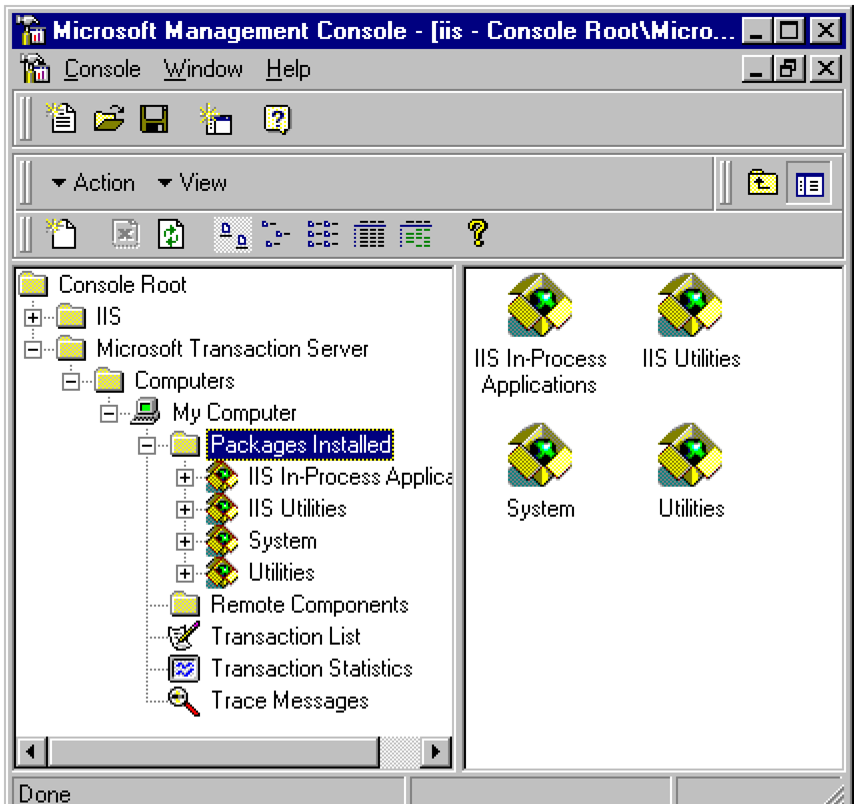
You can push and pull components only if a shared network directory has been established for storage and delivery of DLLs and type library files. (You can choose any shared directory as long as the component files are contained within one of the folders or subfolders of the shared directory.) The MTS Explorer will automatically locate available shared network directories on servers. On a given server you can have multiple shared directories to access different sets of component files.

In addition, you must ensure that:

- You logged on using a Windows NT account that is a member of the Administrators role of the System Package on the target computer.
- The target computer's System Package identity maps to an NT account that is in the Reader role for your System Package.
- Security is enabled for the System package on both computers. See the Enabling MTS Package Security topic for more information.

To push components, you must first add the appropriate computer or computers to your MTS Explorer and then add your components to the remote computer's Remote Components folder. See the Configuring Your MTS Server topic for information on adding a computer to the Explorer.

Then you must add components to the remote computer's Remote Computer folder.



To add components to the remote computer's Remote Components folder

1 Add the remote computer by selecting the Computers folder and clicking **New** in the **Action** menu.

2. Type a name for the computer you want to add, and then click **OK**. If you do not know the name, you can click the **Browse** button to select a computer.
3. In the MTS Explorer, open the Remote Components folder of the computer to which you want to add remote components.
4. On the **Action** menu, click **New**. You can also right-click and select **New** and then **Component** from the right-click menu.
5. In the dialog box that appears, select the remote computer and package that contain the component you want to invoke remotely.
6. From the **Available Components** list, select the component that you want to invoke remotely, and click the down arrow (Add). This adds the component and the computer on which it resides to the **Components to configure on** box. If you click the **Details** checkbox, the Remote Computer, Package, and Path for DLLs are displayed in the **Components to configure on** box.
7. Click **OK**.

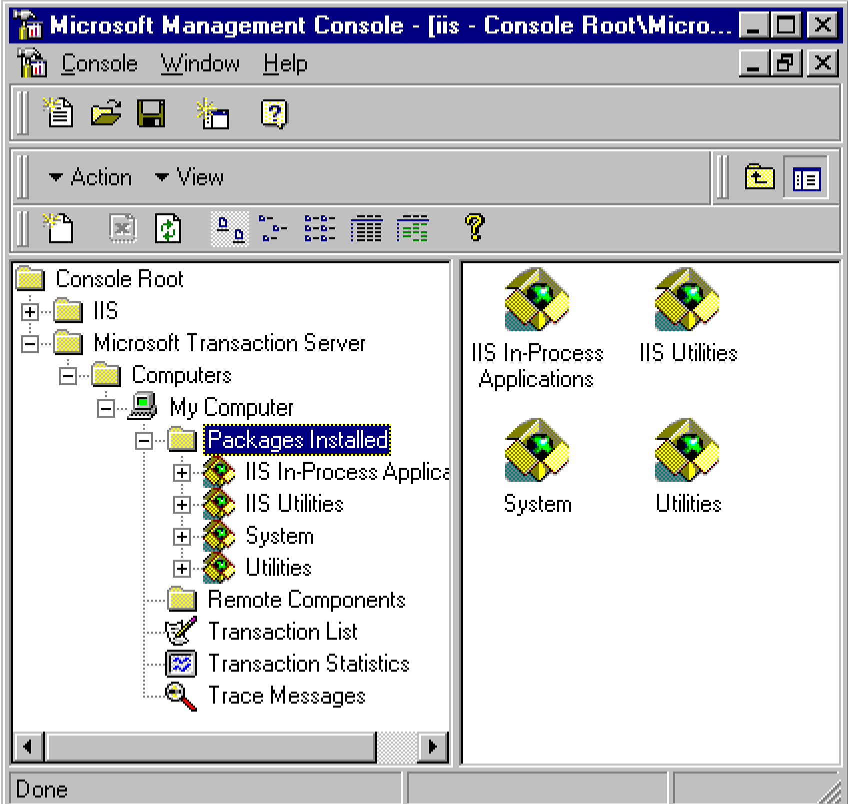
Note When you add components that are stored on a remote computer, the required files are stored in the *install directory*\remote directory. (The default MTS installation directory is \Program Files\MTS.)

See Also

MTS Remote Components Folder

Exporting MTS Packages

Exporting packages allows you to copy a package from one MTS computer to another. For example, you can use the Microsoft Transaction Server Explorer to export a package from a server on which the package was developed to another MTS server for testing.



To export a package:

1 Select the package you want to export in the left pane of the Explorer.

2 On the **Action** menu, click **Export**. You can also right-click and select the **Export** option.

3 In the **Export Package** dialog box, enter or browse for the package file to create. The component files will be copied to the same directory as the package file.

4 If you want to include any roles that you have identified for the package, click the **Save Windows NT user ids associated with roles** checkbox.

5 Click **Export**.

When you export a package, Microsoft Transaction Server creates a package file (with the .pak extension) containing information about the components and roles (if any) included in the original package, and copies the associated component files (dynamic-link libraries (DLL), type libraries, and any proxy/stub DLLs) to the same directory in which the package file was created. Only component DLLs are copied. Package locks against changes or deleting will be exported with the package.

Important If a component's class ID (CLSID), type library identifier (TypeLibId), or interface identifier (IID) change after you have exported a package, you must export the package again.

See Also

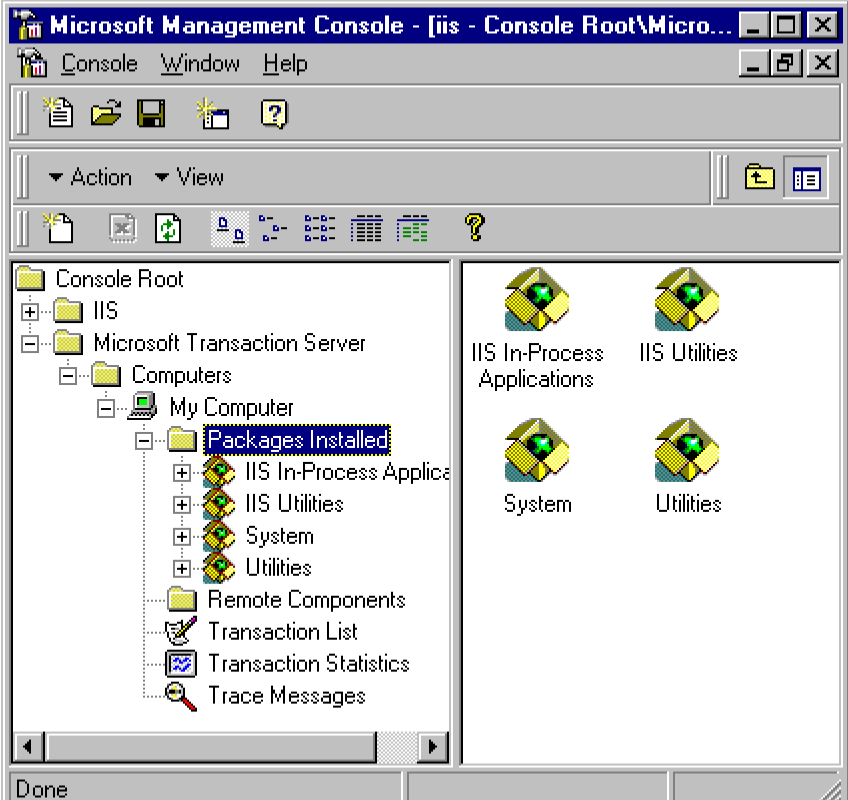
Building an MTS package for Export, Locking Your MTS Package, Mapping MTS Roles to Users and Groups

Generating MTS Executables

Using the MTS Explorer, you can generate an application executable that installs and configures a client computer to access a remote server application. The client computer that installs the executable must have distributed COM (DCOM) support, but does not require any MTS server files other than the executable to access a remote MTS server application.

The application executable utility is part of the exporting packages feature in the MTS Explorer. This utility allows you to automatically generate application executables that install a client application and configure the client computer to access a server application on a remote MTS servers. Executables generated by the application executable utility by default will configure a client computer to access the deployment server on which the executable was generated.

You can also configure the **Options** tab on the Computer property sheet to point client applications to a server other than the deployment server. If you do not enter the name of another server computer before exporting the package to generate an executable, application executables created on the local computer will automatically configure client computers to access server packages on the local computer.



To configure a client application executable to access a server computer other than the local computer:

1. Select **My Computer**.
2. Right-click and select **Properties** from the right-click menu.
3. Select the **Options** tab and enter the name of the remote server in the **Remote server name** field in the Replication section.
4. Click **OK**, and export the package to create the application executable.

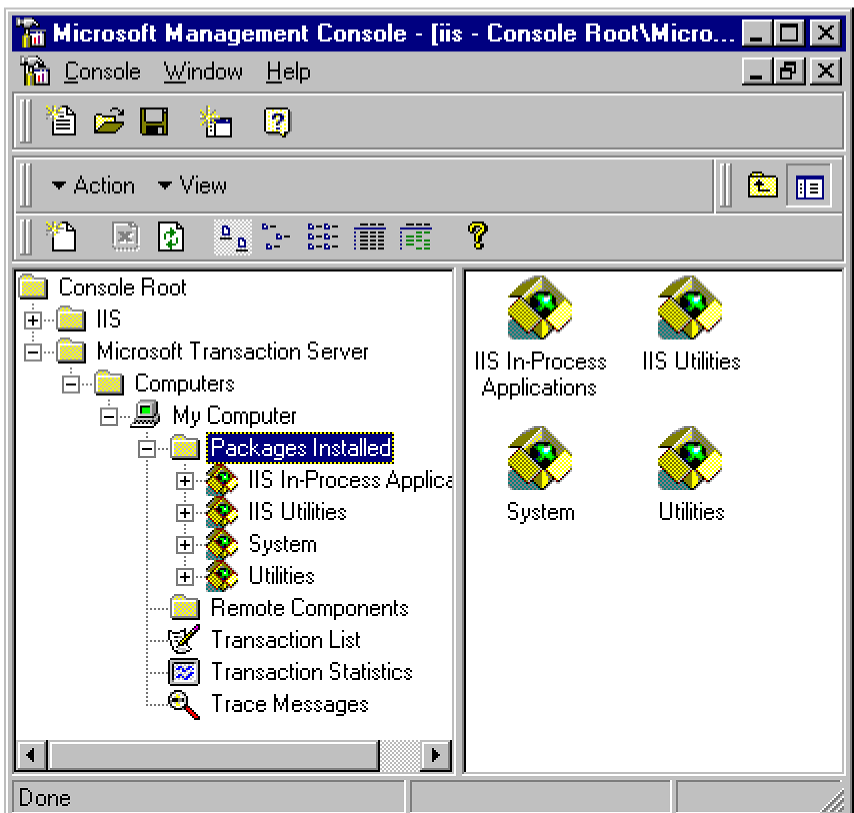
On the MTS server, the application executable utility automatically creates an executable for the client application.

On the client, the executable automates the following steps:

1. Copies to a temporary directory on the client or server machine and then extracts the necessary client-side files, including type libraries and custom proxy-stub DLLs.
1. Transfers type libraries and proxy-stub DLLs for the server application to the Remote Applications directory in the \Program Files sub-directory. All remote applications are stored in the Remote Applications directory. Each remote application has an individual directory named by the package globally unique identifier (GUID).
1. Updates the system registry with the entries that either enable clients to use the server application remotely (including information that is related to application, class, programmatic, interface, and library identifiers) or allows the server application to run on the server computer.
1. Registers the application so that a user can use the **Add/Remove Programs** icon in Control Panel to remove it at a later date. All remote applications are prefaced with "Remote Application" so that you can easily find your application in the list of installed components.
1. Deletes files in the temporary directory generated during installation of the application.

When run on a client computer, the client executable copies the necessary proxy-stub DLLs and type libraries to that computer and updates the client's system registry with information needed by DCOM, including the name of the server computer. Client applications can then access a remote server application.

Before you export a package to create an executable, you must configure your client installation file (clients.ini). You can customize the installation of your client to include additional files, such as client executables, application documentation, or a simple readme. For example, the clients.ini file, which is in the \Clients sub-directory, can be modified to combine installation of client executables for several different applications.



To customize installation:

1 Open the clients.ini file, which is located in the \clients sub-directory.

2 Below the Client Application Files heading, enter the path to the directory in which you want the source code installed on the client computer. For example,

```
Source Path=c:\pgram files\mtx\test\vb bank
```

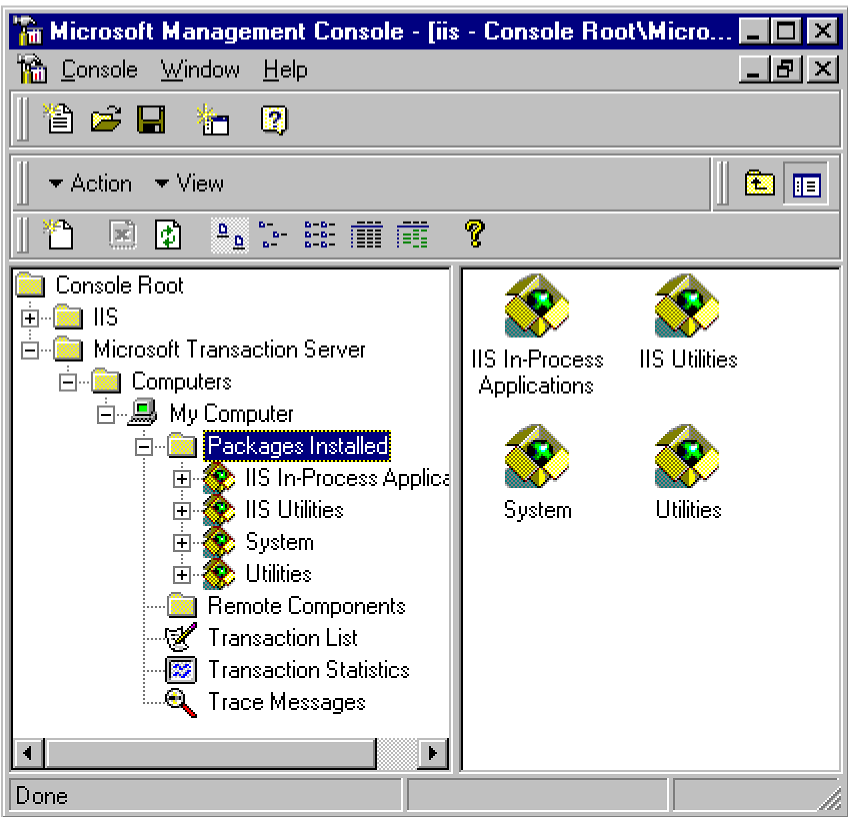
3 Below the ClientApplicationInstallCommands heading, enter the names of the files that you wish to install. Use triple-brackets to enclose the names of the files that you want to install. For example:

```
1=notepad {{{readme.txt}}}
2={{{vbbank.ex}}}}
```

4 Below the ClientApplicationSetup heading, indicate if you would like to enable the ExploreApplication setup option by typing “Y” for yes or “N” for no. For example, the following entry would display the application immediately after setup so that the client could create a shortcut to the desktop:

```
ExploreApplication=Y
```

After you have customized the clients.ini file, you can use the MTS Explorer to export the server package to create a client executable.



To create a client executable:

1 Install the server application using the Package Wizard if you have not already done so.

2 If you would like your clients to access a server other than the deployment server on which you are creating the executable, follow these steps.

- Select **My Computer** in the left pane of the Explorer.
- Right-click and choose the **Properties** option from the right-click menu.
- Click the **Options** tab. In the Remote Server Name box, enter the server name that you want clients to access. For example:

```
\\anotherservermachine
```

1 Export your package from the server on which your server application is installed to another server. Specify a new filename (*YourNewFileName*) as the package export filename, and place the exported package in your MTS directory.

2 Locate the folder into which you exported your package. You will see a Clients subdirectory that contains a single file named *YourNewFileName*.exe. Exporting an existing package in the MTS Explorer generates a "Clients" subdirectory beneath the directory to which you exported the package. The Clients sub-directory contains a single executable with the name specified during

package export. When run on any client supporting DCOM, this executable installs all the necessary information for remote clients to access the server application.

Important Do not run this client executable file on your server computer. Running the client executable on the server computer removes the registry entries required to run the server package. If you make this mistake, you must remove the application using the **Add/Remove Programs** property sheets in the Control Panel. Then delete and re-install the package using the MTS Explorer.

Distributing the Client Executable

The MTS Explorer automates packaging and installing client applications into executables for distribution. Distribute those executables by using one of the following methods:

- Sharing a directory so that clients can copy the executable and run it on their computers.
- Sending an email attachment so that clients can save and run the executable on their computers.
- Incorporating an executable into an HTML script using the <OBJECT> tag. The <OBJECT> tag allows the browser to download the application from a specified object store location to the client machine if the client initiates an event on the HTML page (such as clicking a button). Using the <OBJECT> tag to distribute executables facilitates upgrading executables, since the browser automatically checks the client's registry for the current version of the application. If the existing executable is outdated, the browser downloads the latest version from the object store.
- Using Microsoft System Management Server (SMS) to "push" the distribution of your application from a central location to tens or hundreds of computers at once. Note that you must install the client application itself after installing the application on remote computers.

Removing the Client Executable

Clients can remote the client executable through the **Add**/**Remove** option in the Control Panel. Applications installed by MTS executables begin with "Remote Application" in the Install/Uninstall list. To remove the executable, select the appropriate application and click **Remove**.

See Also

Exporting MTS Packages, Building an MTS Package for Export

Installing MTS Packages

Installing MTS packages consists of installing and deploying packages in the MTS run-time environment using the MTS Explorer. Installation and deployment procedures are closely linked with design considerations, such as setting the appropriate package identity.

When you deploy applications, make sure that you thoroughly understand the design of the package and its components. Refer to the *MTS Programmer's Guide* for more information about designing and building MTS components.

You can easily deploy pre-built packages using the MTS Explorer. This section discusses the following topics:

Installing Pre-built MTS Packages

Upgrading MTS Packages

Enabling MTS Package Security

Setting MTS Package Identity

Adding a New MTS Role

Mapping MTS Roles to Users and Groups

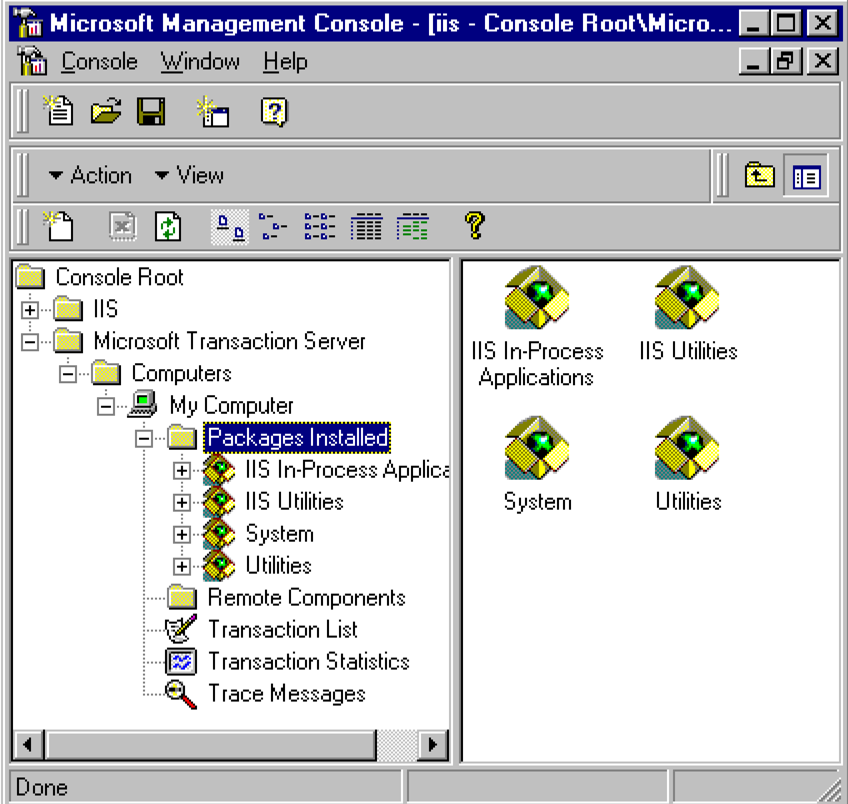
See Also

Identity Tab (Package), Packages Installed Folder, Roles Folder, Users Folder, Role Membership Folder, Managing Users for MTS Roles

Installing Pre-built MTS Packages

A pre-built package consists of a package file and the component files (.dll and .tlb) associated with the package. Use the MTS Explorer to install and deploy pre-built packages developed by third party or enterprise developers.

Note that after you install any package, the *minimum* configuration for your package includes controlling access to objects stored by MTS, mapping users to roles, and setting security levels for packages. See the Enabling MTS Package Security topic for more information on how to configure security for your package.



To install a pre-built package:

1. In the left pane of the Explorer, select the computer on which you want to create a package.
2. Open the **Packages Installed** folder for that computer.
3. On the **Action** menu, click **New**. You can also select the **Create new object** button on the MTS toolbar, or right-click on the **Packages Installed** folder and select **New** and then **Package**.
4. Click the **Install pre-built packages** button.
5. In the **Select Package Files** dialog box, click **Add** to browse the network for available package files. Select a package file (.pak), click **Open**, and then click **Next**. Note that you can install multiple packages at the same time. The component files that are included in the package must be located in the same directory as the package file.
6. Specify the package identity in the **Set Package Identity** dialog box, and then click **Next**. The default selection is **Interactive User**. The interactive user is the user that logged on to the Windows NT account for the computer that runs a package. You can select a different user by selecting the **This user** option and entering details for a specific Windows NT user or group.
7. In the Installation Options dialog box, specify the installation directory. The component files are copied from the package file directory to the installation directory. You can accept the default directory, or click **Browse** to search for another location.
8. If the package file you are installing contains defined Windows NT users, the **Add Windows NT**

users saved in the Package File option will be available. Clicking this box adds these users to the new package.

9 Click **Finish**. The Explorer hierarchy now shows the new package in the right pane. If you install multiple packages, the options you choose in steps 6 and 7 apply to all the packages.

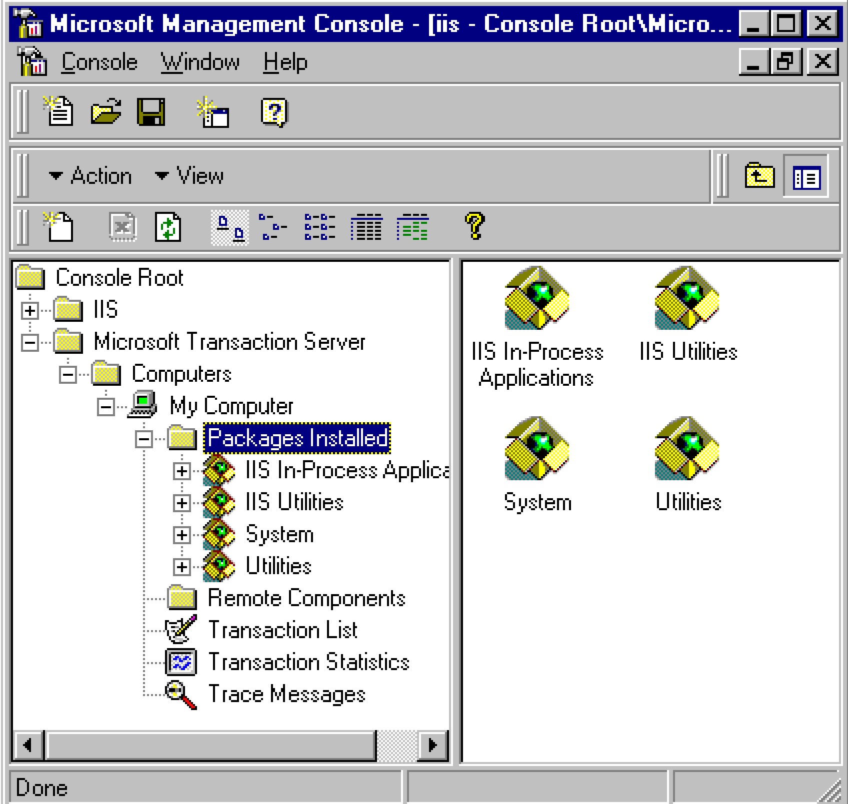
On Windows NT, you can also install a pre-built package by dragging a package file from the Windows NT Explorer to the right pane of the MTS Explorer while the **Packages** folder of a computer is open.

See Also

Packages Installed Folder

Upgrading MTS Packages

If you want to upgrade an MTS package, you must first delete the previous version and then install the updated package.



To delete a package:

1. In the left pane of the Explorer, select the computer for which you want to delete a package.
2. Open the **Packages Installed** folder for the specified computer to display all the packages.
3. Select the package you want to remove.
4. On the **Action** menu, click **Delete**. You can also right-click and select the **Delete** option from the right-click menu.
5. Click **Yes** to remove the package.

Deleting the package also deletes any components contained in the package. You can also delete a package by selecting it and pressing the DELETE key.

Refer to the Installing Pre-built MTS Packages topic for instructions on installing the upgraded package.

See Also

Packages Installed Folder

Enabling MTS Package Security

MTS offers two types of package security:

- Programmatic security

  Provides interfaces that you can use to create customized security within your application logic. See the *MTS Programmer's Guide* for more information about using programmatic security.
- Declarative security

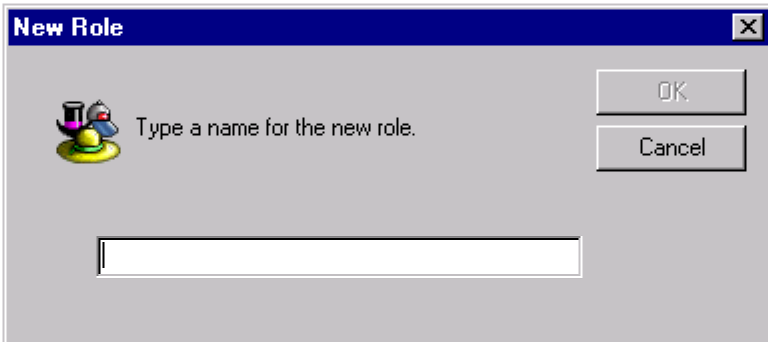
  Allows you to define roles and assign Windows NT users or groups of users to roles using the MTS Explorer.

Important Library package do not support role checking. In order to enable security, you must change the activation setting to a server package. See the Setting MTS Activation Properties topic for more information about library and server packages.

Administrators use declarative security to secure packages, ensuring that only clients with access privileges can run the package. Access is granted through the Explorer using MTS roles and Windows NT-based user and group accounts. Note that since declarative security uses Windows NT accounts for authentication, you will not be able to use declarative security for a package running on a Windows 95 computer.

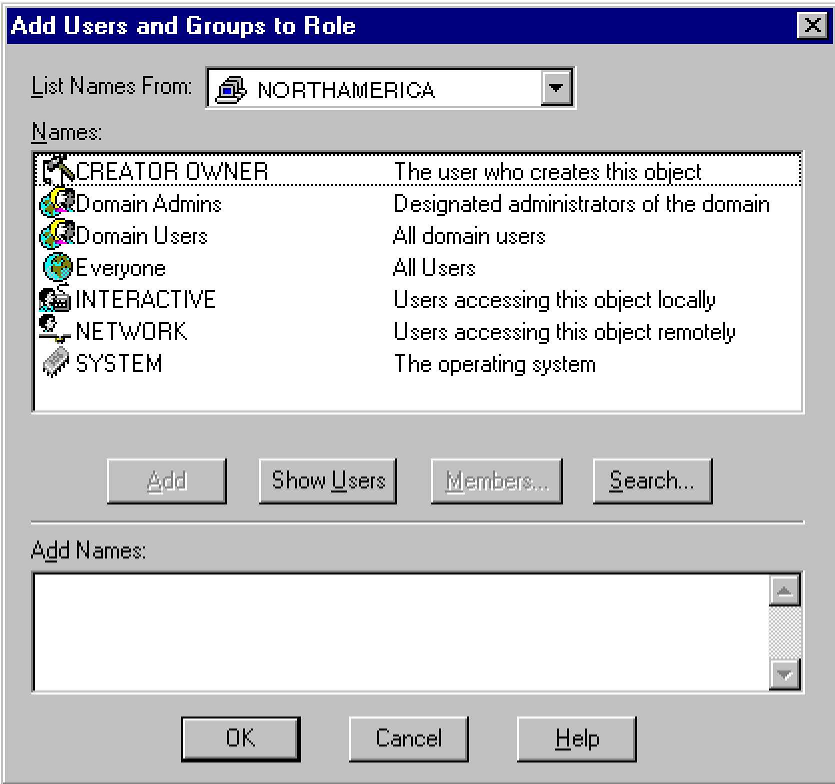
To set up declarative security for a package, perform the following steps:

1. Define roles at the package level using the **New Role** dialog box.



   See the Adding a New MTS Role topic for a description of how to add a new role.

1. Map users to roles using the Add New Users to Roles dialog box. Note that a package with no valid users in any Role cannot be called.



See the Mapping MTS Roles to Users and Groups topic to learn how to add users and groups to a role.

1. Assign the role that you defined to the Role Membership folder of a component or interface if you want to restrict access to a specific component or interface.
2. Enable security for the package on the **Security** tab of the Package property sheets. This topic contains a description for how to enable authorization checking.

If you do not map the user account you're currently using to the Administrator role before enabling System package security, you will be refused access to MTS Explorer functions that modify configuration (such as adding users to roles). If this happens, you need to log on as a user that has been mapped to the Administrator role. To protect administrators from being locked out of the System package, the MTS Explorer displays an error message if you try to:

- Enable security for the System package when no users are mapped to the administrator role
- Delete the last user from the Administrator role when security has been enabled for the System package

Note If MTS is installed on a server whose role is a primary or backup domain controller, a user must be a domain administrator in order to manage packages in the MTS Explorer.

If you do not enable security for the package, then roles for the component or interface will not be checked by MTS. In addition, if you do not have security enabled for a component, MTS will not check roles for the component's interface.

See the Adding a New MTS Role topic for a description of how to assign a role to the Role Membership folder.

Note Turning off declarative security for individual components or the package is useful during debugging of your package.

Consider setting up access restrictions to an inventory server package. As the system administrator, you may want to restrict access to the Inventory package to members of the sales department. In

order to do so, first select the Role folder for the Inventory package, click the **New** option on the **Action** menu, and type "Sales" as the name of the new role. Then select the Users folder, click **New** on the **Action** menu, and enter the name of the Windows NT group account for the sales department. Add the Sales role to each component's Role Membership folder. At this point, only members of the sales department are allowed to access the Inventory package. Finally, select that package, go to the Security tab of the property sheets, and select the **Enable authorization checking** checkbox in order to turn on the new security settings for the package.

If you want to restrict access to a specific component within a package, you must understand how components in the package call one another. If a component is directly called by a base client, MTS will check roles for the component. If one component calls another component in the same package, MTS will not check roles because components within the same package are assumed to "trust" one another.

Let's say that you wanted to configure roles to permit a client to call the CheckInventory component, and restrict the client from calling the Backorder component directly. Both the CheckInventory and Backorder components are in the Inventory package. You must first set the appropriate role on the CheckInventory component for the client. Then ensure that the Backorder component has no roles that could map to the client identity. Since the CheckInventory and Backorder components share a package, no role checking will be performed when the CheckInventory component calls the Backorder component.

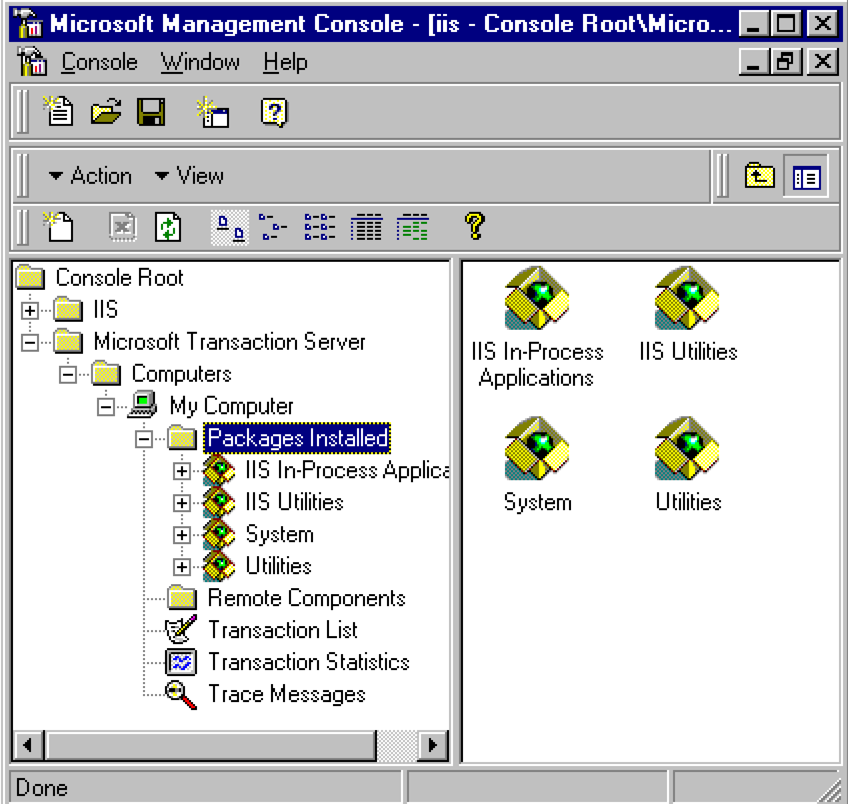
The CheckInventory component may call the Backorder component on behalf of the client, though, if the following conditions are fulfilled:

- The client identity maps to the appropriate CheckInventory role
- Any programmatic security requirements are satisfied.

This allows you to create packages containing mutually trusted components while restricting access to select components.

To set up role checking for original callers that directly call the Backorder component, select the Role Membership folder for the Backorder component, click **New** on the **Action** menu, and choose the **Sales** role. Now that the **Sales** role (with mapped users) is assigned to the Backorder component, only members of the sales department will be able to run the Backorder component to view out-of-stock items. To activate the new security setting, select the **Enable authorization checking** checkbox for the Inventory package as well as the Backorder component.

For more information about role checking, see the Programmatic Security topic in the *MTS Programmer's Guide*.



To enable security authorization:

1. Map your user account to the **Administrator** role of the System Package if you have not already done so.
2. Select the System Package, and choose **Properties** from the **Action** or right-click menu.
3. Go to the security tab and select the **Enable authorization checking** checkbox.
4. Stop the System Package server process by selecting System Package, right-clicking, and choosing the **Shut Down** option.

   You can also shut down all server packages at one time, which combines steps 4 and 7. To shut down all server packages, select **My Computer** and choose the **Shut Down Server Processes** option in the **Action** menu.
5. Select the package for which you want to enable security.
6. Go to the security tab and select the **Authorization checking enabled** checkbox.
7. Stop the System Package server process by selecting that package, right-clicking, and choosing the **Shut Down** option.

After you install and configure your package on the deployment server, you may want to lock your package so that component configurations cannot be modified. Refer to the Locking Your Package topic for more information about locking your package configuration.

See Also

System Package, Roles Folder, Users Folder, Role Membership Folder, Managing Users for MTS Roles, Microsoft Transaction Server Programmer's Guide

Setting MTS Package Identity

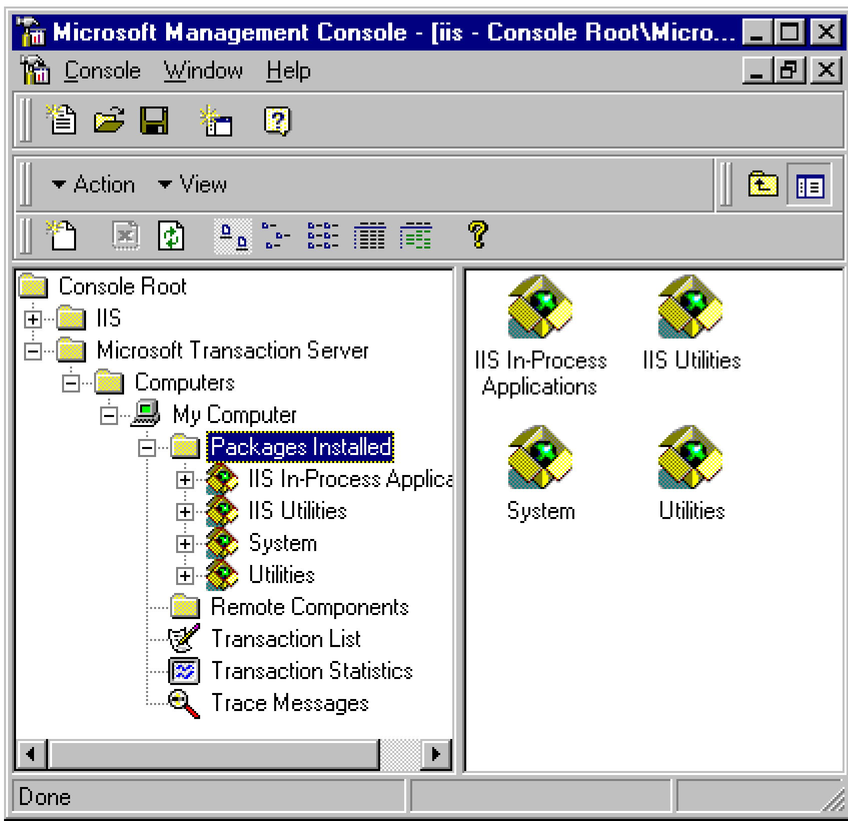
You can configure a package (which is a single server process) to run as one of the following package identities:

- Interactive User
- A specified Windows NT user account

By default, MTS packages run as Interactive User.

In many deployment scenarios, it is preferable to run a package as an Windows NT user account. If a package runs as a single Windows NT user account, you can configure database access for that account rather than for every client that uses the package. Permitting access to accounts rather than individual clients improves the scalability of your application.

For example, consider an Accounting package that updates a SQL Server database with billing and sales information. You can configure the database Accounting table to allow read access for users in the Windows NT accounting clerk group account. You can then set the package identity to the accounting clerk group account, which allows members of that account to run the package and read data from the Accounting table.



To set package identity to a specified user account:

1. Select the package whose identity you want to change.
2. On the **Action** menu, click **Properties** and select the **Identity** tab.
3. Select the **This user** option and enter the user domain followed by a backslash (\), user name, and password for the Windows NT user account.

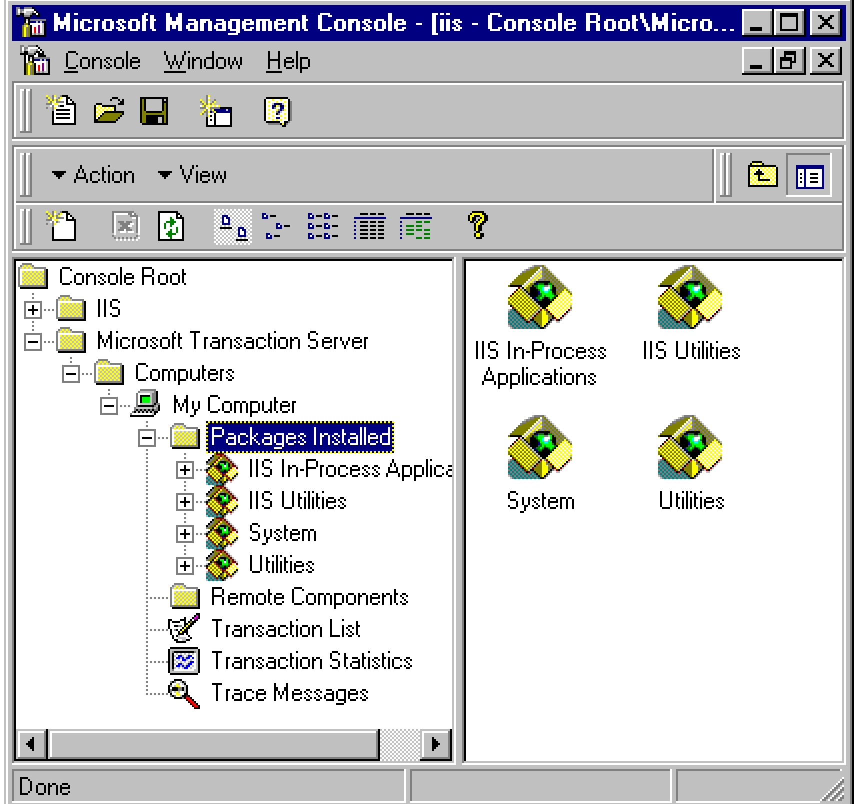
   Note that if you want to use package identity to restrict access to a database, you must set database access privileges for the user account.

See Also

Mapping MTS Roles to Users and Groups, Enabling MTS Package Security, Adding a New MTS Role, Identity Tab (Package)

Adding a New MTS Role

Although new roles are usually added during package development, you may have to add a new role to an existing package. Roles represent a set of system-level privileges that are required for a particular business function. Roles are set at the package level. You can use the MTS Explorer to map Windows NT users or groups of users to the roles that you create.



To create a new role:

1 In the left pane of the Explorer, select the package that will include the role.

2 Open the **Roles** folder.

3 On the **Action** menu and click **New**. You can also select the **Roles** folder and click the **Create new object** button, or right-click the **Roles** folder and select **New** and then **Role**.

4 In the dialog box that appears, type the name of the new role.

5 Click **OK**.

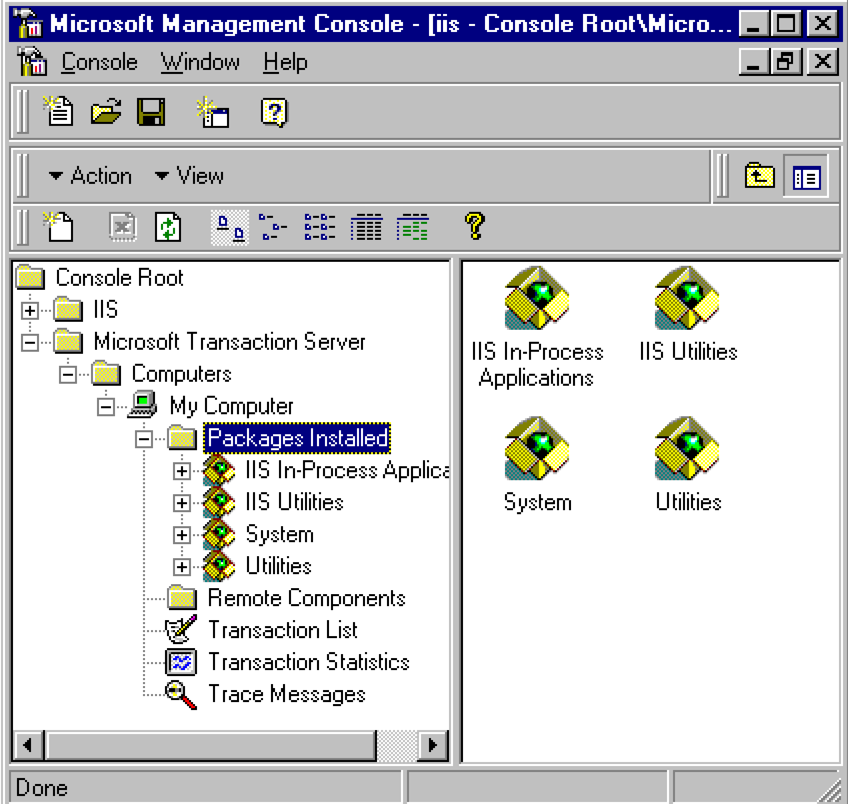
Caution Package security will not be enabled unless you map a valid user to a package role.

See Also

Mapping MTS Roles to Users and Groups, Enabling MTS Package Security, Setting MTS Package Identity, Roles Folder, Managing Users for MTS Roles

Mapping MTS Roles to Users and Groups

When you install and deploy your application, you must map Windows NT users and groups to any existing roles. Roles determine user access for components and interfaces.



To assign users to roles:

1. In the left pane of the Explorer, select the package that contains the component to which you want to assign roles.
2. Open the **Roles** folder.
3. Double-click the role to which you want to assign users.
4. Open the **Users** folder.
5. On the **Action** menu, click **New**. You can also select the **Users** folder and click the **Create new object** button, or right-click the **Users** folder and select **New** and then **Users**.
6. In the dialog box that appears, add user names or groups to the role. You can use the **Show Users** and **Search** buttons to locate a user account.
7. Click **OK**.

See Also

Adding a New MTS Role, Enabling MTS Package Security, Setting MTS Package Identity, Roles Folder, Users Folder, Managing Users for MTS Roles

Maintaining MTS Packages

After a package has been successfully built, distributed, and deployed, administrators can use the MTS Explorer to maintain MTS packages. Maintaining MTS packages entails monitoring the status and properties of packages and their components, and reconfiguring package and component properties if required.

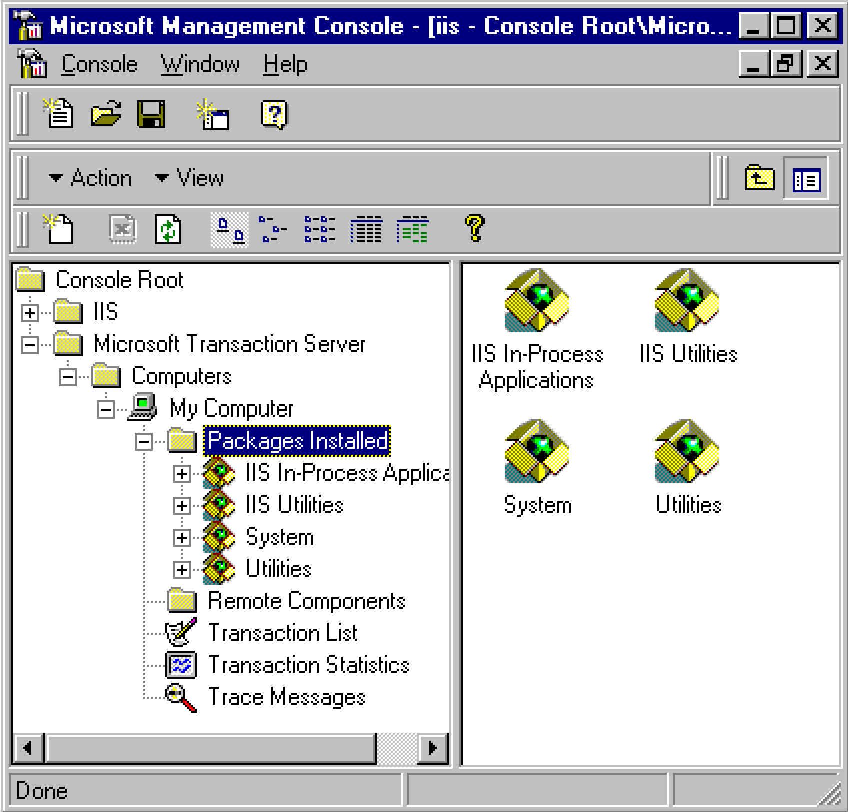
This section discusses the following topics:

Monitoring Status and Properties in the MTS Explorer
Using Property Sheets in the MTS Explorer
Managing Users for MTS Roles
Using MTS Replication

Monitoring Status and Properties in the MTS Explorer

You can use the MTS Explorer **Status View** and **Property View** settings to monitor the status and properties of active packages and their components. You can view the status or properties of items in the Explorer by selecting the item and then clicking the **Status View** or **Property View** button in the MTS Explorer toolbar.

The **Status View** lets you see the status of computers, packages, and components in the Explorer. For example, **Status View** for the Computers folder indicates if Microsoft Distributed Transaction Coordinator (MS DTC) has been started, while **Status View** for the Packages Installed folder indicates which package is running. **Status View** for components in a package allows you to see the component's programmatic identifier (progID), objects activated, and the objects in call. You can use the **Status View** to quickly see the status of computers and application processes that you are managing.



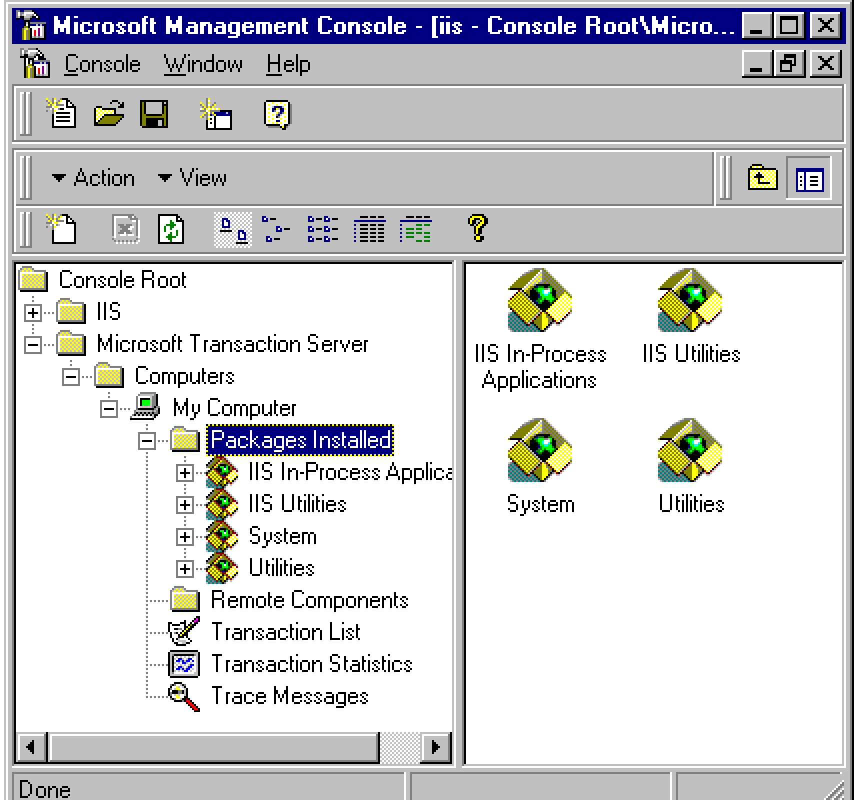
To use the Status View in the MTS Explorer

1 Select the **Packages Installed** folder or **Components** folder.

2 Choose the **Status View** icon on the MTS toolbar in the right pane of the MTS Explorer.

Note **Status View** does not display the status of the Utilities and System packages.

Use the **Property View** to quickly gather information about the properties of packages and components. For example, the property view for components displays the component's ProgID, transaction setting, dynamic-link library (DLL) location, class ID (CLSID), threading model, and security authorization settings. For example, you can use the **Property View** to quickly view the transaction properties for all components in a package.



To use Property View in the MTS Explorer

1. In the right pane of the MTS Explorer, select the folder whose status you would like to view, such as the **Packages Installed** or **Components** folder
2. Click the **Property View** button on the MTS Explorer toolbar. The properties for items in that folder will be displayed in columns in the right pane of the MTS Explorer.

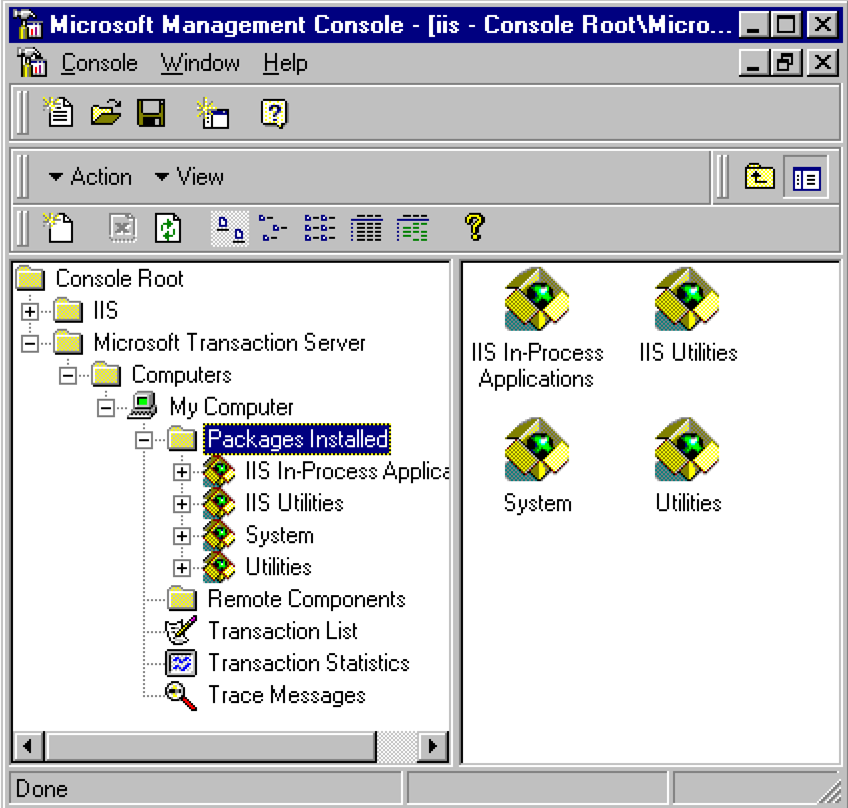
See Also

Using MTS Property Sheets

Using Property Sheets in the MTS Explorer

Package and component property sheets allow you to change the configuration of MTS packages. For example, if you would like to change the transaction property for a component, you can modify the component property sheet for transaction settings. Before you re-configure a specific property setting, review the Creating MTS Packages and Installing MTS Packages topics for descriptions and procedures on setting different properties in the MTS Explorer.

Packages can be locked against modification or deletion of the package and its components. If a package has been locked during installation or deployment, you will have to unlock the package before modifying component properties. For more information about locking, see Locking Your MTS Package .



To configure package and application properties using property pages

1. Right-click the item whose properties you want to set. You can also select the item in the left pane of the MTS Explorer and choose **Properties** from the **Action** menu.
2. Modify the appropriate property sheet and click either **OK** or **Apply**.
3. Refresh all components of My Computer by selecting the My Computer icon and choosing **Refresh all components** from the **Action** menu. You can also refresh individual packages by selecting the Packages Installed folder and clicking the **Refresh** icon on the MTS toolbar in the right pane of the MTS Explorer.

   Note If you change the activation level of your package, you have to shut down that package process before the change will take effect. You can shut down the package process by right-clicking the package and choosing the **Shut down** option.

See Also

Monitoring Status and Properties in the MTS Explorer

Managing Users for MTS Roles

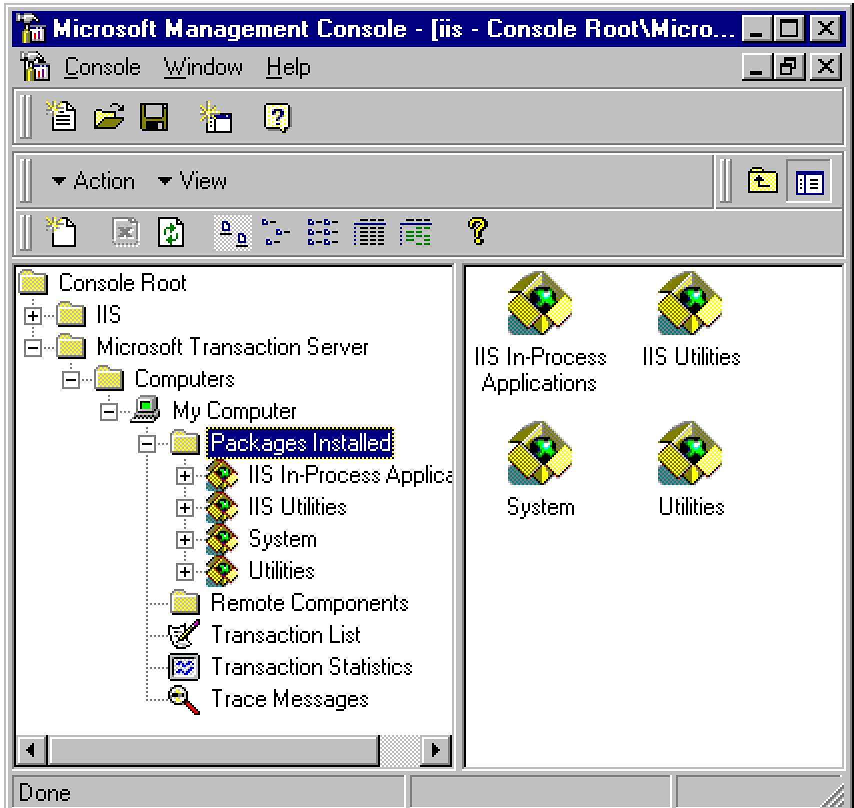
As an administrator, you will have manage a role's Windows NT users and groups of users by:

- Adding a new user or group to a role
- Removing a user or group from a role
- Moving a user or group from one role to another

As clients of your server packages increase, use the MTS Explorer to map the new users to roles for the package. For more information about mapping new users to a role, refer to the Mapping MTS Roles to Users and Groups topic.

You may have to remove users from a role, or move users or groups of users from an existing role to a newly created role. For example, if an employee leaves the company, you would remove that user from the roles with which the former employee was associated.

When you create new roles for an existing package, you may have to move a user or a group of users from one role to another. For example, if you create a Clerk role for an Accounting package, you may have to move some users or a group of users from the existing Manager role (with read and write privileges for Accounting data) to the Clerk role (with read-only privileges). In order to move users or groups of users, you must remove the user or group of users from the role and then map them to the new role.



To remove a user from a role

1. Locate the package that contains the role from which you want to remove a group or a user. In the left pane of the MTS Explorer, select that package.
2. Open the **Roles** folder.
3. Double-click the role that has been defined for the user you want to remove.
4. Open the **Users** folder.
5. Select the user you want to remove.

6. On the **Action** menu, click **Delete**. You can also right-click and select **Delete** from the right-click menu.
7. Click **Yes** in the dialog box that appears. You can also remove a user by selecting the icon for that user and pressing **Delete**.

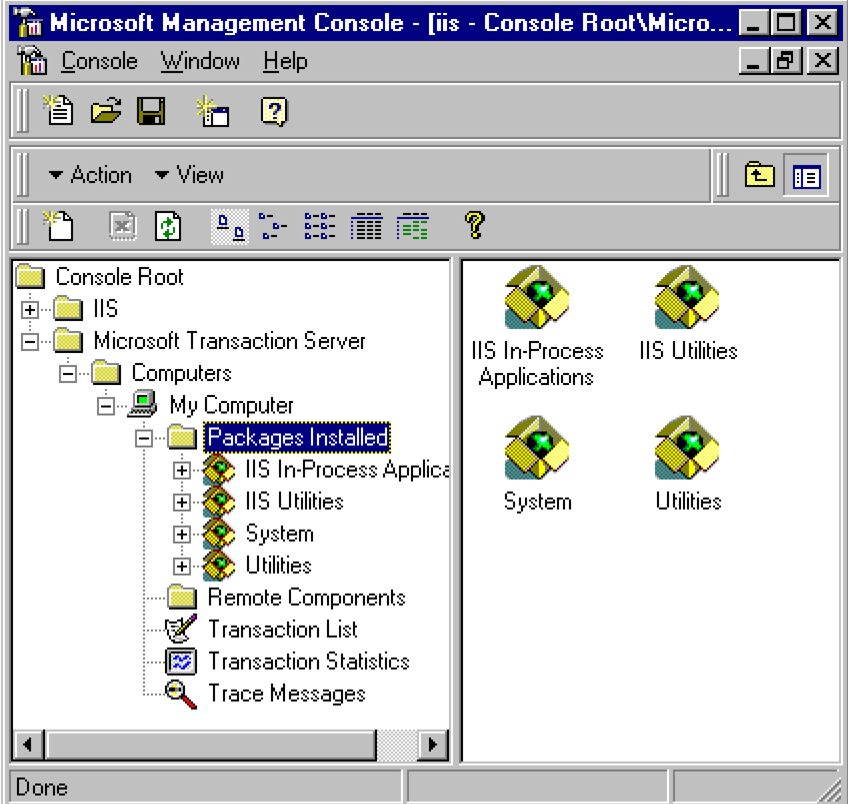
See Also

Roles Folder, Users Folder, Enabling MTS Package Security, Setting MTS Package Identity, Adding a New MTS Role, Mapping MTS Roles to Users and Groups

Using MTS Replication

MTS provides replication for use with Microsoft Cluster Server (MSCS). If an MTS server in a cluster fails or is taken offline, the other server takes over the failed server's operations.

How to Replicate an MTS Server



To replicate an MTS server

1. Install MTS on both computers. For more information, see Configuring MTS with Microsoft Cluster Server.
2. Designate a master computer and install your packages on that computer.
3. Register any imported components on both computers.
4. Ensure dependencies such as runtime libraries are installed on both computers. (See the Limitations section in this topic.)
5. Specify a **Replication Share** name for the master on the **Options** tab on the **My Computer** property sheet (see the Package Contents Replication section in this topic).
6. Run the MTXREPL.EXE command-line utility.

You use the MTXREPL.EXE command-line utility to replicate an MTS server. Both the master and destination computers must be running. MTXREPL.EXE takes the following arguments:

```
MTXREPL.EXE master destination
```

MTS replication uses a single-master approach, copying MTS catalog information from a master computer to a destination computer. To replicate to another destination computer, run the MTXREPL.EXE utility again. The single-master approach is not strictly enforced, however, so you can in turn use the destination computer as the master for another MTS server replication.

During replication, MTS *completely* replaces the catalog on the destination with the catalog on the source, except where noted below in the What MTS Replication Does Not Do section in this topic.

You cannot do partial replication. If MTS replication fails, run the MTXREPL.EXE utility again.

Administrators are responsible for initiating MTS replication and must ensure that package changes to the master server are replicated to all computers within the cluster.

It is recommended that replication be performed when the destination is not currently running MTS components. There is a window of failure on MTS startup while reading catalog information. If the replication were to occur at this time, package startup will fail, requiring the client to retry.

Replicating IIS Packages

Use the Microsoft Internet Information Server (IIS) version 4.0 replication utility to replicate IIS packages. IIS replication copies all MTS catalog information automatically. Because of this, running the MTS replication utility is not necessary; in fact, it will not run if IIS 4.0 is installed.

Package Contents Replication

MTS replication copies all necessary component dynamic-link libraries (DLL)s, type type libraries, and proxy-stub DLLs. To allow destination computers to access these files, you must specify a **Replication Share** name for the master on the **Options** tab on the **My Computer** property sheet. You must create this share on the master and grant read-only privileges to destination computers. The **Replication Share** name is the share name, not the directory that the share name maps to. For example, if you have d:\mtx\repl shared as MyReplPoint, then **Replication Share** should be MyReplPoint, not d:\mtx\repl.

The **Replication Share** must grant read-only permission to the identity of the destination computer's System package. If the destination's System package performs role checking, then the identity of the System package on the master must belong to the Administrator role in the destination's System package.

The location of the first component DLL in a package determines where files are installed on the destination computer. If this component is located in a subdirectory beneath the default package directory, the subdirectory is mirrored on the destination computer. The default package directory is determined during MTS setup; for example, if you install MTS into c:\program files\mtx, then the default package install directory is c:\program files\mtx\packages.

If your packages are installed beneath this directory, they will be installed in the same location relative to the default package install directory on the destination computer. For example:

Computer	Default Package Directory	Package Install Directory
Master	c:\program files\mtx\packages	c:\program files\mtx\packages\MyPak
Destination	d:\mts\packages	d:\mts\packages\MyPak

Packages installed outside the default package install directory are installed in the "external" subdirectory on the destination computer. For the above example:

Computer	Package Install Directory
Master	Package A in c:\program files\mtx\packages\MyPak Package B in d:\misc
Destination	Package A and B in d:\mts\packages\MyPak

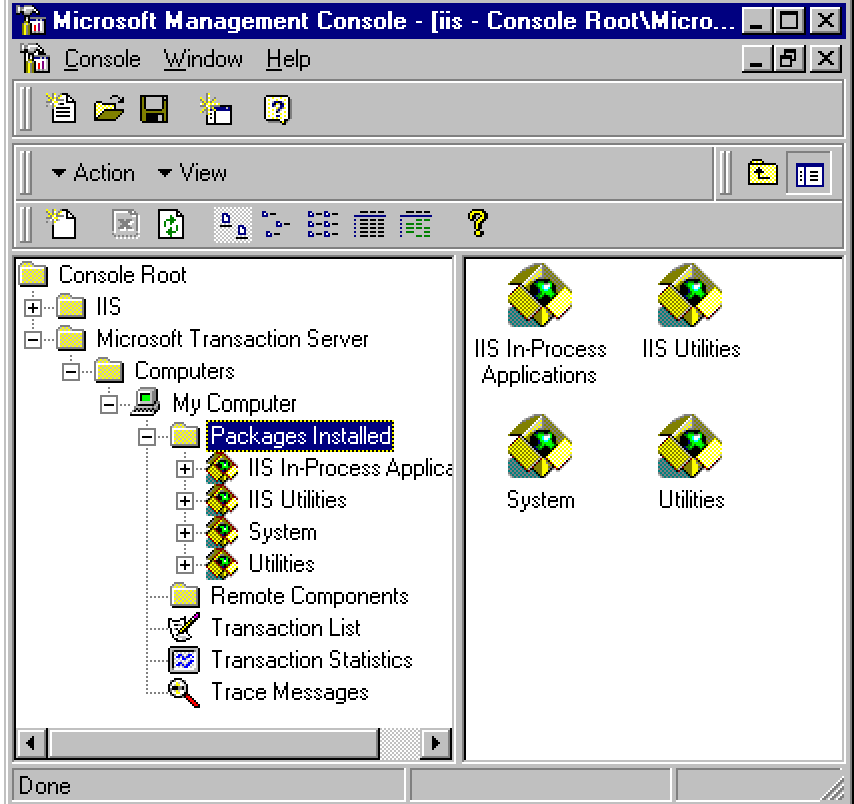
It is recommended you install packages beneath the default package directory. It is also recommended that all files of the same package reside in the same directory.

Administering MTS Server Clusters

Administrators must always use physical computer names for administration in an MSCS cluster. If you are on the master computer in an MSCS cluster and you want to view the copy, use its physical computer name. To replicate between the master and copy, use the physical computer names.

You must specify the virtual server name in the **Remote Server Name** on the **Options** tab on the **My Computer** property sheet for deploying MTS applications on an MSCS cluster. When you export a package using the application executable utility, clients which then install the application will be directed to the virtual server. If you do not set the **Remote Server Name**, clients will be directed to the physical computer, which is not the proper configuration for failover protection.

You can also perform static load balancing with failover protection by specifying different virtual server names. By having two virtual server names, one for each physical computer, you can create a client installation executable for a package installed on both computers. By dividing their distribution among clients, static load balancing is achieved.



To load balance an MTS application using virtual servers

1. Decide on how you want to balance load.

   For example, you can run all packages on only one computer at a time, run all packages on both computers, or run some packages on one computer and some on the other.
2. Use the MSCS Cluster Administrator to create virtual server names for both computers.
3. Export packages using the virtual server names that you created in Step 2.

   If you want to run all packages on only a single computer at a time, export all packages under a single virtual server name. If you want to run all packages on both computers at the same time, export all packages twice, one for each virtual server name You must then distribute the two different sets of client installation executables to among your clients. If you want to run half the packages on one computer and half the packages on the other, export each half using a different virtual server name.
4. Run the client executables on the client machines. Depending on how you chose to load balance in Step 1, you may have several executables to run on each client.

What MTS Replication Does Not Do

MTS replication does not do the following:

- Install MTS. You must setup MTS on each destination computer before running MTS replication.
- Replace MTS system and utilities packages and the IIS utilities packages. You must configure these packages identically on the master and destination (although file paths can differ). MTS does not enforce this; it is the responsibility of the administrator to synchronize both sets of packages as needed.
- Replicate computer-specific information such as the **Replication Share** and **Remote Server Name** properties.
- Replicate MTS component dependencies that cannot be detected by MTS such as run-time libraries. You must copy these files on each destination computer in order for applications to execute properly.
- Replicate local computer security accounts. Such accounts are not recommended for use within clusters for roles or package identities.
- Replicate persisted data that your application relies on. For example, SQL Server provides automatic failover for its databases; you must ensure that the database is configured to failover for your application.

Limitations

- MTS replication to or from Windows 95 computers is not supported.
- MTS replication will fail if there are any remote components on the master which are set to run on the destination computer.
- Imported components can only be replicated if they are already registered on the destination computers. Installed components are automatically registered and configured on the copy by the replication utility. When replicating components that were imported on the master, the replication utility expects to find the components already registered on the copy. It is your responsibly to manually register imported components on each destination computer prior to running the replication utility.
- Replication to or from versions of MTS prior to MTS 2.0 is not supported.

See Also

Configuring MTS with Microsoft Cluster Server, Options Tab (Computer)

Managing MTS Transactions

Understanding how distributed transactions work helps you effectively manage transactions for Microsoft Transaction Server (MTS) packages. This section describes how transactions work, and how to monitor and manage transactions in the MTS Explorer.

This section discusses the following topics:

Understanding MTS Transactions
Managing MS DTC
Monitoring MTS Transactions
Monitoring MTS Transactions with Windows 95
Understanding MTS Transaction States
Resolving MTS Transactions

See Also

Transaction List, Transaction Statistics, Transaction Icons, Trace Messages

Understanding MTS Transactions

Microsoft Transaction Server (MTS) enables you to easily use, monitor, and administer distributed transactions in your applications. A distributed transaction is a transaction involving updates to transaction-protected resources on more than one system. The MTS transaction manager, Microsoft Distributed Transaction Coordinator (MS DTC), provides a distributed transaction facility for Windows NT and Windows 95 systems. MS DTC also makes it possible to update two or more transaction-protected resources on a single system.

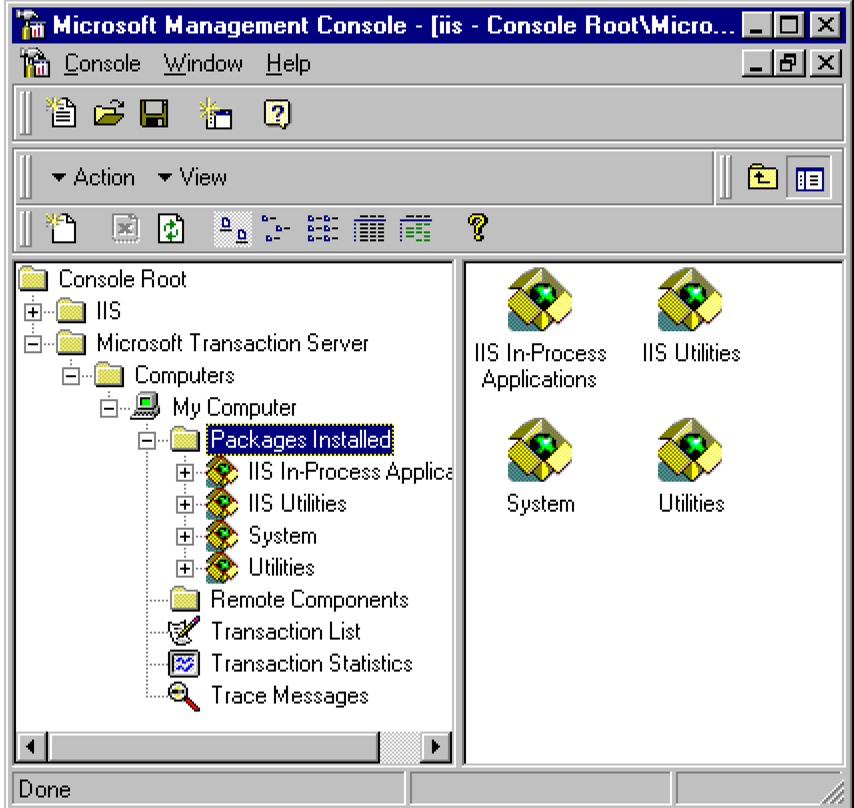
See Also

Managing MS DTC, Monitoring MTS Transactions, Monitoring MTS Transactions with Windows 95, Understanding MTS Transaction States, Resolving MTS Transactions

Managing MS DTC

In order to manage transactions, you must first start Microsoft Distributed Transaction Coordinator (MS DTC). If you have not started MS DTC on your MTS server, clients cannot access the transactional packages that you are managing.

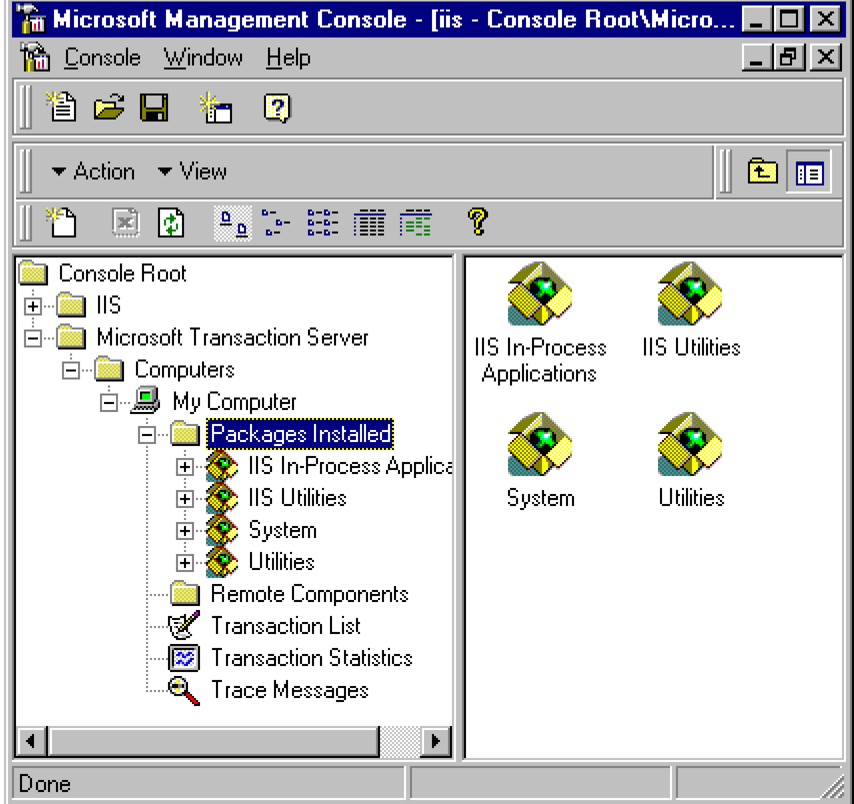
To configure and run MS DTC clients on a Windows NT machine, the user must have write access to the local registry and remote read access to the server's Software\Classes key.



To start or stop MS DTC:

1. In the right pane of the MTS Explorer, select the computer on which you are managing transactional packages.
2. Open the **Action** menu and select **Start MS DTC** or **Stop MS DTC**. You can also right-click and select **Start MS DTC** or **Stop MS DTC** in the right-click menu.

You can also configure DTC settings, such as the location and size of the DTC log file, using the **Advanced** tab of the **Computer** property sheets.



To configure MS DTC settings:

1 In the right pane of the MTS Explorer, select the computer on which you are managing transactional packages.

2 Open the **Action** menu and select **Properties**. You can also right-click and select the **Properties** option from the right-click menu.

3 Select the **Advanced** tab. You can adjust how transactions are displayed in the Transaction List and Trace Message windows. You can also change the location or size of the MS DTC log. Increasing the size of the MS DTC log lets you run more concurrent transactions.

Important MS DTC has an upper limit on the size of the log file:

- On Windows NT, the maximum log size is 512 MB.
- On Windows 95, the maximum log size is 64 MB.

Do Not Compress the MS DTC Log File

The MS DTC log file cannot be compressed.

Removing DTCXATM.LOG Files

Before upgrading your installation of MTS 2.0, delete the DTCXATM.LOG file. Note that you must stop the MS DTC service before deleting DTCXATM.LOG.

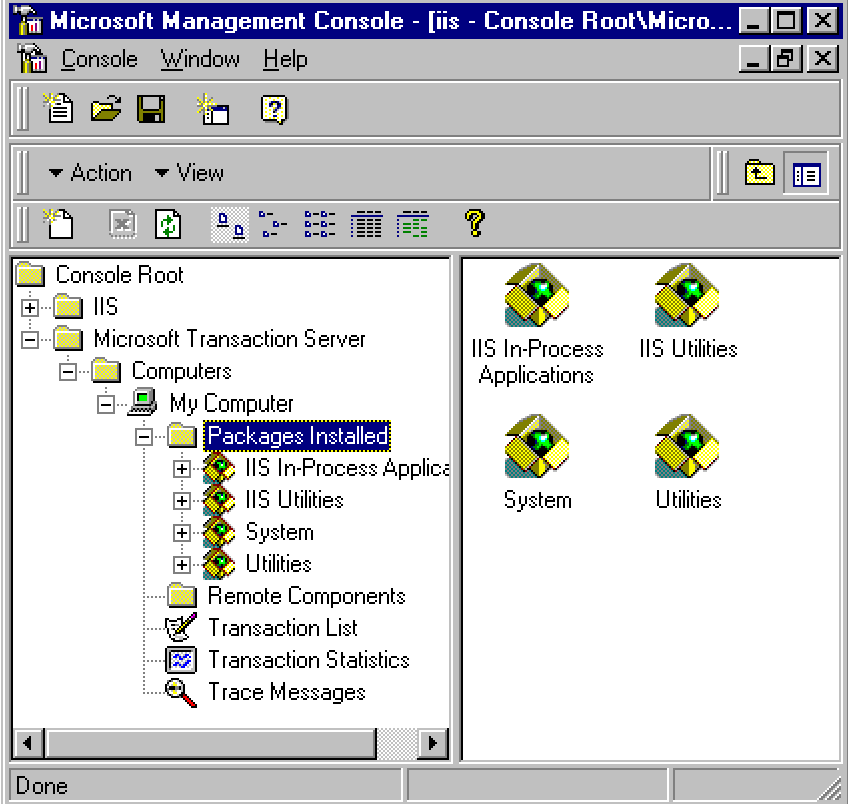
See Also

Understanding MTS Transactions, Monitoring MTS Transactions, Monitoring MTS Transactions with Windows 95, Advanced Tab (MTS Computer), Transaction List, Transaction Statistics, Transaction Icons, Trace Messages

Monitoring MTS Transactions

You can administer transactions in MTS applications by using the transaction windows in the MTS Explorer. The Trace Message, Transaction Statistics, and Transaction List windows provide valuable information about the status of transactions managed by the Microsoft Distributed Transaction Coordinator (MS DTC).

You can use the Transaction List window to resolve transaction states. For more information, see the Resolving MTS Transactions topic.



To monitor transactions using the Transaction List

1 In the left pane of the MTS Explorer, select the computer that is hosting the transactions.

2 Double-click the Transaction List icon.

3 Right-click anywhere in the right pane, and point to the **View** command on the shortcut menu.

4 On the **View** submenu, click one of the following:

- **Large Icon**

  Displays transactions as large icons.

- **Small Icon**

  Displays transactions as small icons.

- **Properties**

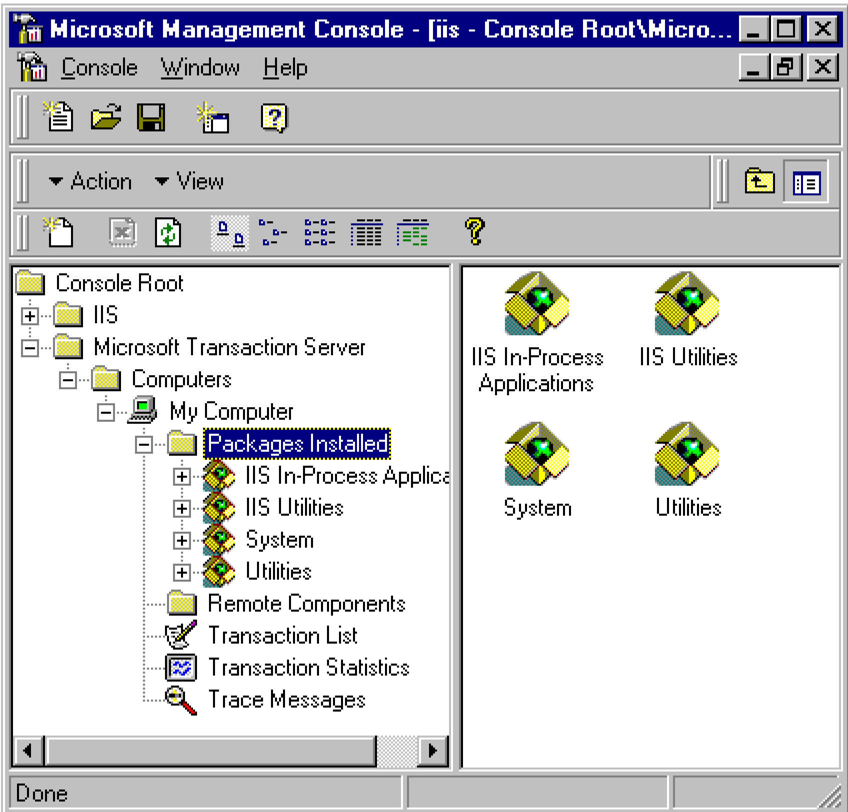
  Lists transactions in a single column and provides a column that contains the unit-of-work ID associated with each transaction. This is the transaction's global unique identifier, and is generated by MS DTC when the transaction begins. Also listed are the parent and subordinate transactions.

- **List**

  Displays transactions sequentially in a column.

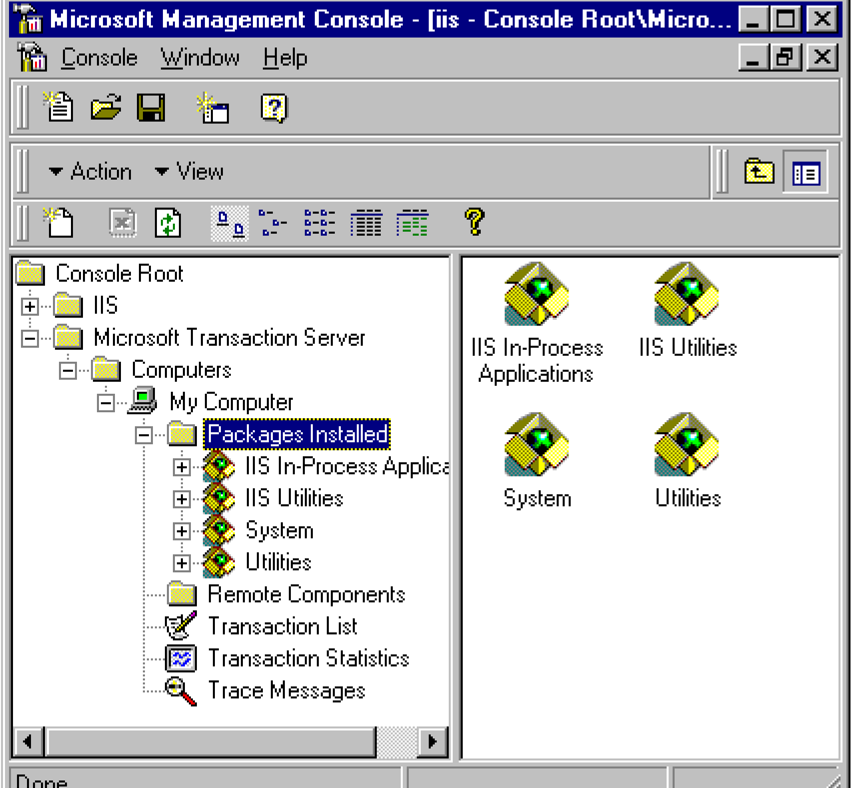
Note The **Small Icon** and **List** commands display the most transactions at any one time.

Properties provides the most information about the transactions, and **Large Icon** displays transactions in the most readable format.



To view transaction statistics

1. In the left pane of the MTS Explorer, select the computer where you want to view transaction statistics.
2. Double-click the Transaction Statistics icon.



To view trace messages

1 In the left pane of the MTS Explorer, select the computer that is hosting the transactions.
2 Double-click the Trace Messages icon.

The MS DTC Transaction Statistics window is displayed in the right pane.

Transaction Properties

You can view transaction properties by selecting the **Properties** command after right-clicking a selected transaction. The **Properties** command lists all transaction managers that are involved in the transaction. For a child transaction, the **Properties** command lists the immediate parent of transactions. If the transaction is a parent transaction, then the immediate child transaction(s) are listed.

See Also

Understanding MTS Transactions, Monitoring MTS Transactions with Windows 95, Understanding MTS Transaction States, Resolving MTS Transactions, Transaction List, Transaction Statistics, Transaction Icons, Trace Messages

Monitoring MTS Transactions on Windows 95

You can manage MTS transactions on Windows 95 or Windows NT computers using the following MTS Explorer windows:

- Transaction Statistics
- Transaction List
- Trace Message.

By default, the Microsoft Distributed Transaction Coordinator (MS DTC) is configured to start automatically when a Windows NT or Windows 95 system starts. To prevent MS DTC from automatically starting after rebooting a Windows 95 computer, use the Registry Editor to find the `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices` registry key and then delete the registry value named `MSDTC`. If you want to enable automatic startup of MS DTC again, use the Registry Editor to create a registry value named `MSDTC` with the value `msdtc-start` under the `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices` registry key .

Note To configure and run MS DTC clients on a Windows NT machine, the user must have write access to the local registry and remote read access to the server's Software\Classes key.

If you want to administer a Windows NT computer remotely from a Windows 95 computer, you must install the Remote Registry service for Windows 95. The Remote Registry service allows you to change registry entries for a remote Windows NT computer (given the appropriate permissions). To obtain the Remote Registry service, go to the \Admin\Nettols\Remotereg sub-directory on the Windows 95 CD. Review the Regserv.txt file for instructions on installing the Remote Registry service.

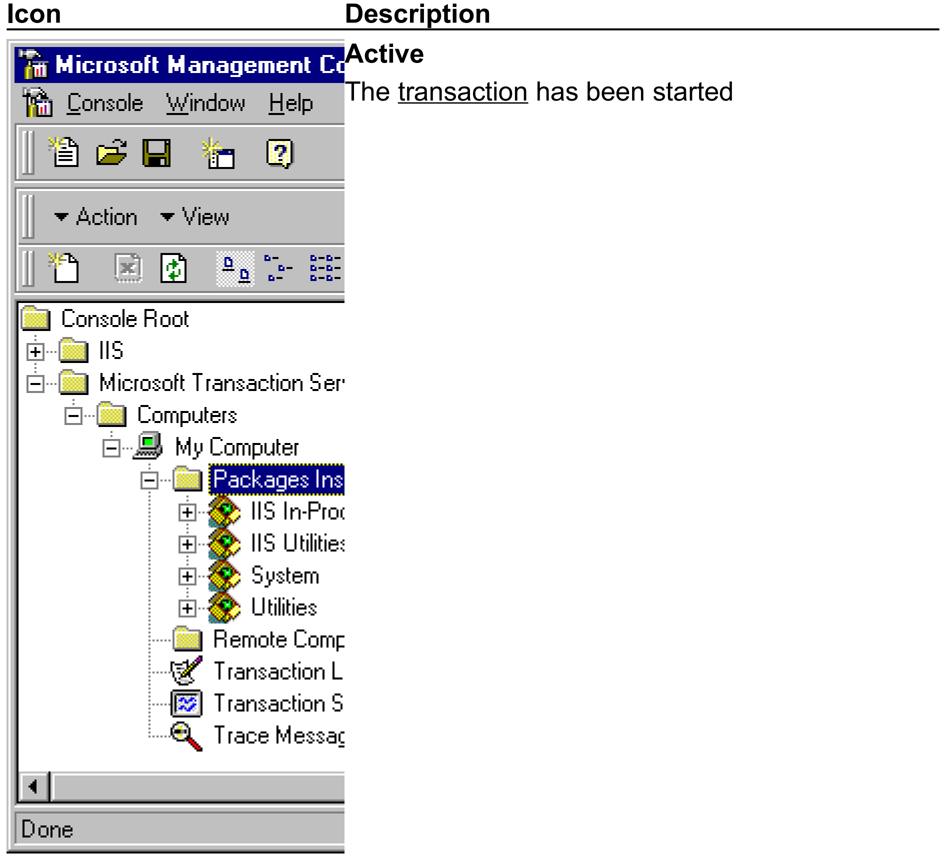
Since Windows 95 does not have an event log, MS DTC events are stored in a text file named msdtc.txt. The msdtc.txt file is located in the \MTSLogs sub-directory of the Windows directory, and contains information about MS DTC events.

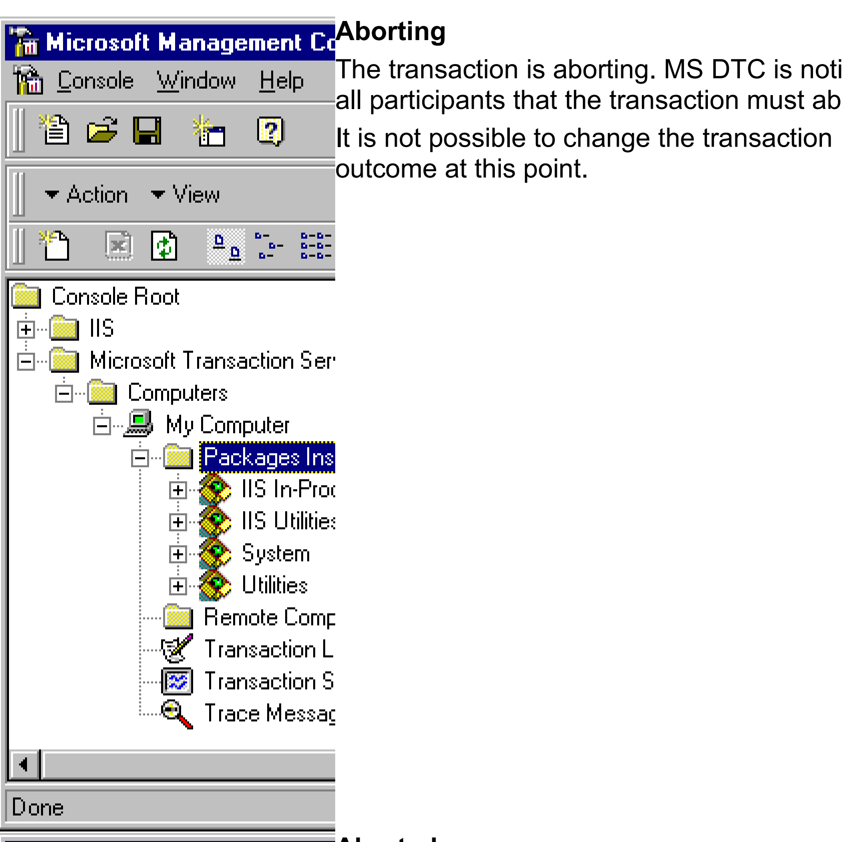
See Also

Understanding MTS Transactions, Monitoring MTS Transactions, MTS Transaction List, Transaction Statistics, Transaction Icons, Trace Messages

Understanding MTS Transaction States

In order to manage transactions, you must understand the different transaction states and their implications for the MTS package that you are administering. Transaction states are represented by the following icons in the Large Icon view of the Microsoft Distributed Transaction Coordinator (MS DTC) Transaction List window:

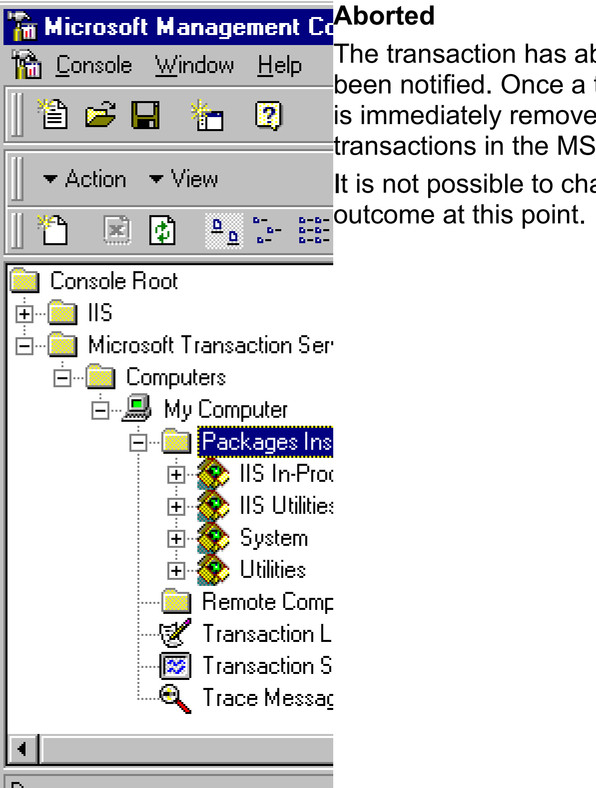
Icon	Description
	**Active** The transaction has been started



Aborting

The transaction is aborting. MS DTC is notifying all participants that the transaction must abort.

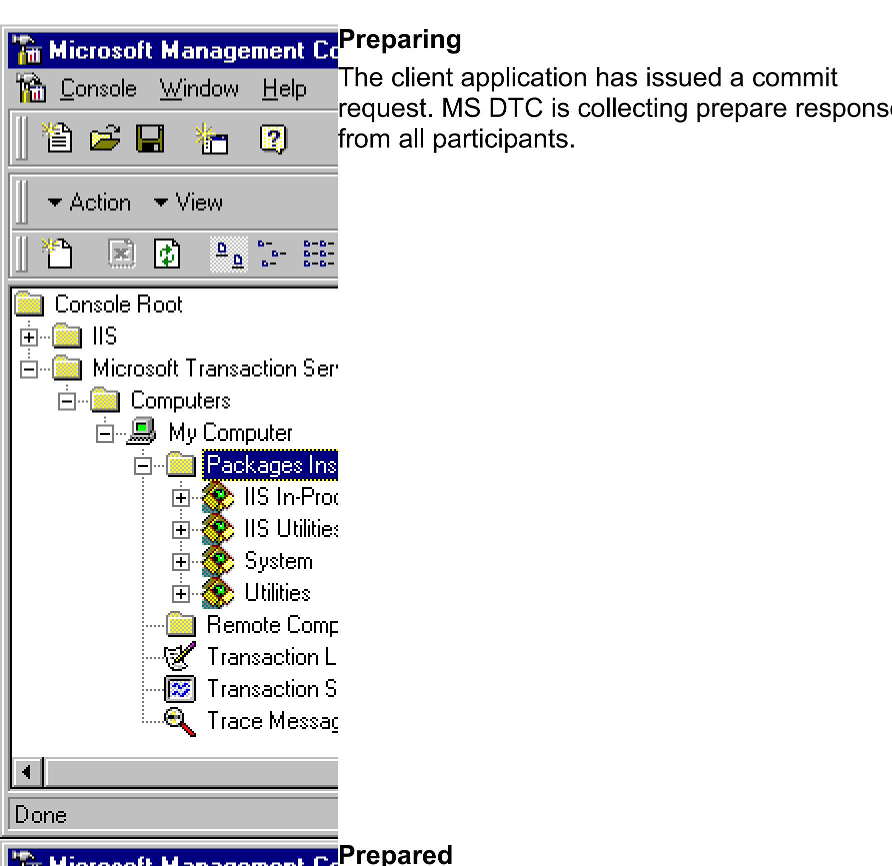
It is not possible to change the transaction outcome at this point.



Aborted

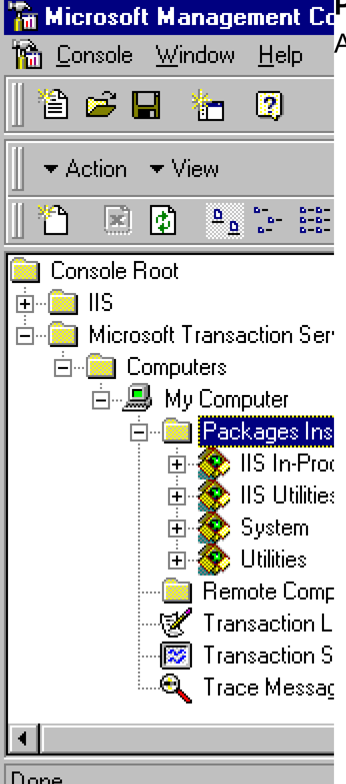
The transaction has aborted. All participants have been notified. Once a transaction has aborted, it is immediately removed from the list of transactions in the MS DTC Transactions window.

It is not possible to change the transaction outcome at this point.



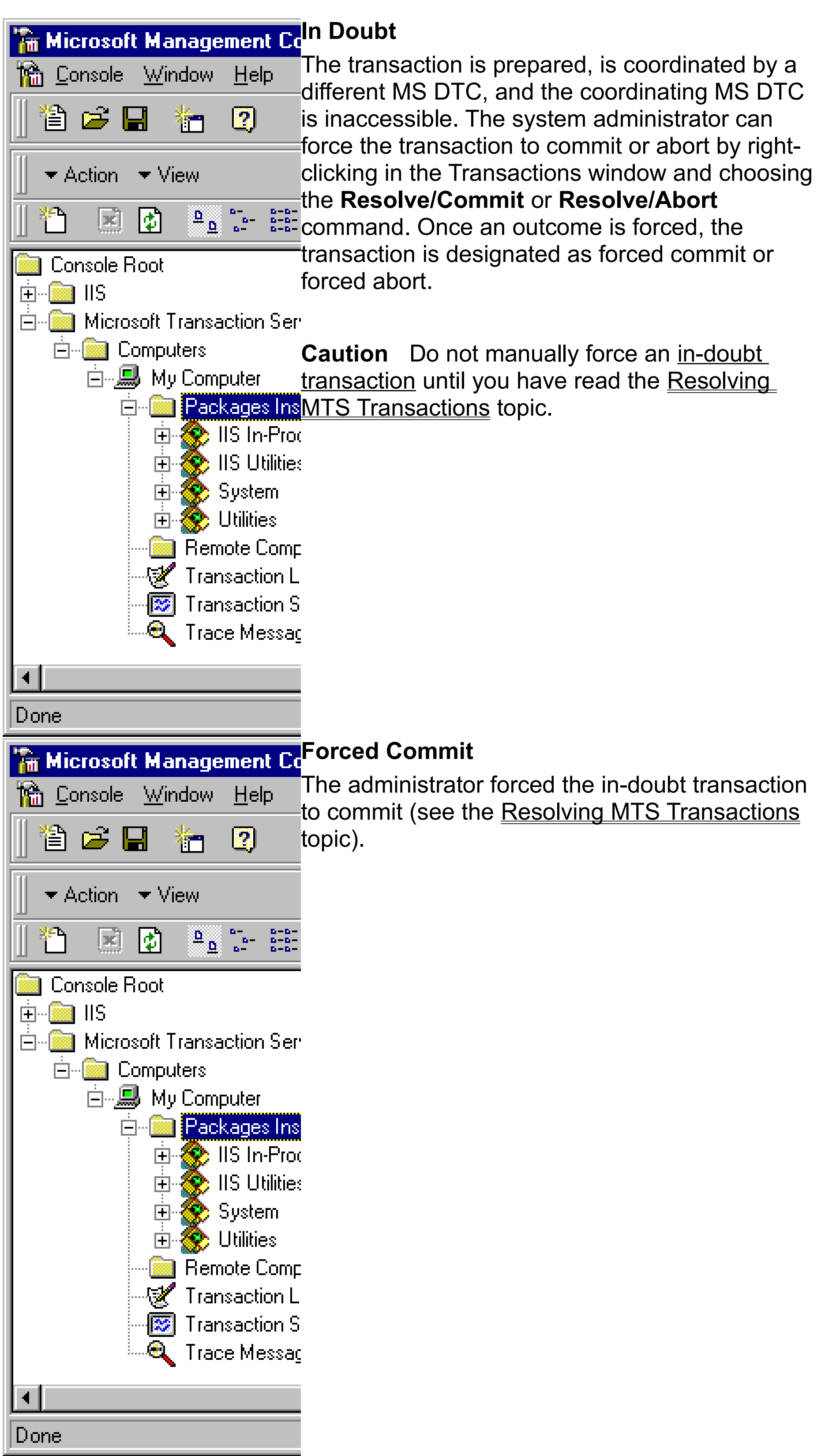
Preparing

The client application has issued a commit request. MS DTC is collecting prepare responses from all participants.



Prepared

All participants have responded yes to prepare.



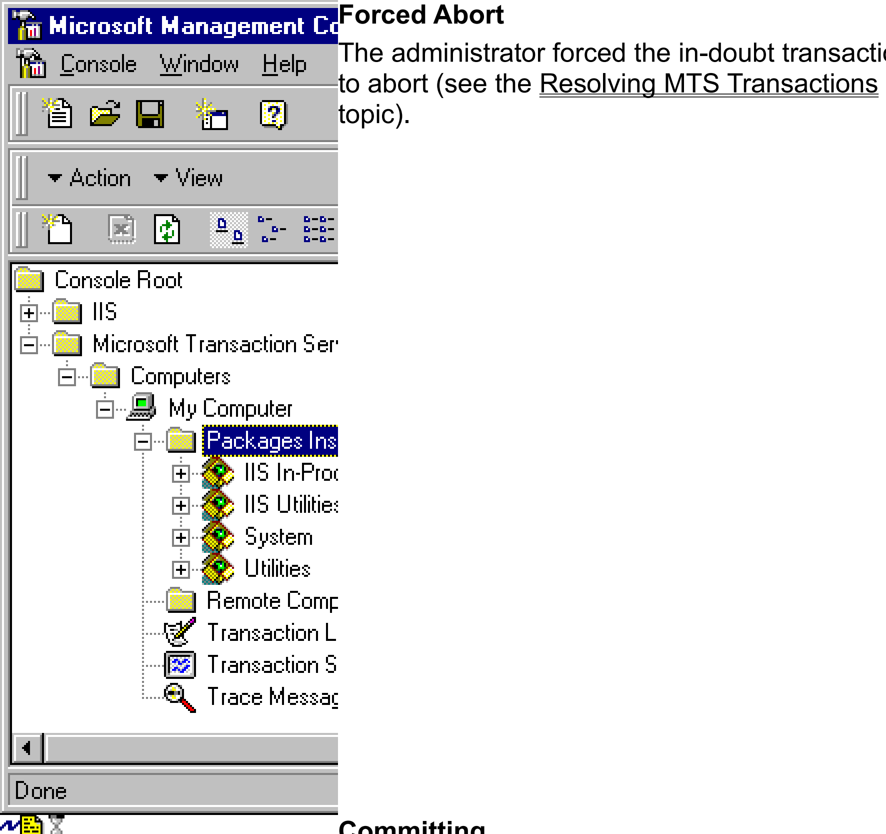
In Doubt

The transaction is prepared, is coordinated by a different MS DTC, and the coordinating MS DTC is inaccessible. The system administrator can force the transaction to commit or abort by right-clicking in the Transactions window and choosing the **Resolve/Commit** or **Resolve/Abort** command. Once an outcome is forced, the transaction is designated as forced commit or forced abort.

Caution Do not manually force an in-doubt transaction until you have read the Resolving MTS Transactions topic.

Forced Commit

The administrator forced the in-doubt transaction to commit (see the Resolving MTS Transactions topic).



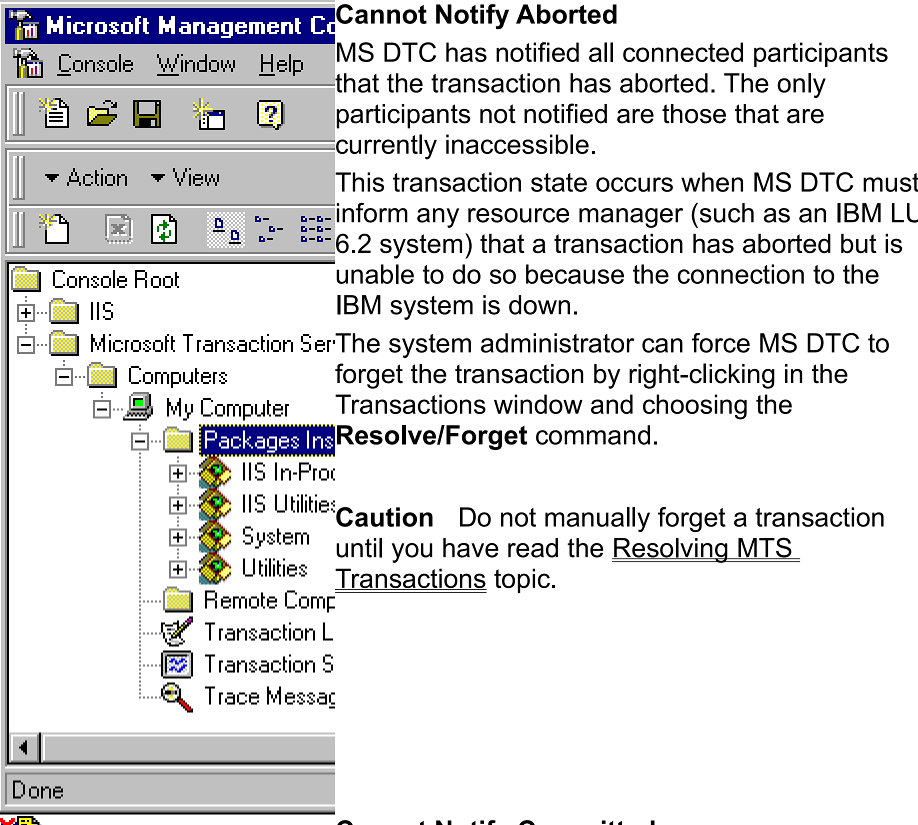
Forced Abort

The administrator forced the in-doubt transaction to abort (see the Resolving MTS Transactions topic).

Committing

The transaction has prepared successfully and MS DTC is notifying participants that the transaction has been committed. MS DTC does not end the transaction until all participants have acknowledged receiving (and logging) the commit request.

It is not possible to change the transaction outcome at this point.



Cannot Notify Aborted

MS DTC has notified all connected participants that the transaction has aborted. The only participants not notified are those that are currently inaccessible.

This transaction state occurs when MS DTC must inform any resource manager (such as an IBM LU 6.2 system) that a transaction has aborted but is unable to do so because the connection to the IBM system is down.

The system administrator can force MS DTC to forget the transaction by right-clicking in the Transactions window and choosing the **Resolve/Forget** command.

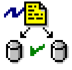
Caution Do not manually forget a transaction until you have read the Resolving MTS Transactions topic.

Cannot Notify Committed

MS DTC has notified all connected participants that the transaction has committed. The only participants not notified are those that are currently inaccessible.

The system administrator can force MS DTC to forget the transaction by right-clicking in the Transactions window and choosing the **Resolve/Forget** command.

Caution Do not manually forget a transaction until you have read the Resolving MTS Transactions topic.



Committed

The transaction has committed and all participants have been notified. Once a transaction commits, it is immediately removed from the list of transactions in the MS DTC Transactions window.

It is not possible to change the transaction outcome at this point.

See Also

Understanding MTS Transactions, Monitoring MTS Transactions, Monitoring MTS Transactions with Windows 95, Resolving MTS Transactions, Transaction List

Resolving MTS Transactions

If you administer MTS transactions, you may have to manually resolve a transaction for an MTS application. You can use the Transaction List window in the MTS Explorer to resolve a transaction by choosing one of the following commands:

- **Commit**   This command forces the transaction to commit.
- **Abort**   This command forces the transaction to abort and roll back to its original state.
- **Forget**   This command deletes a committed or aborted transaction from the Microsoft Distributed Transaction Coordinator (MS DTC) log. You should always force an outcome for a transaction before using this command.

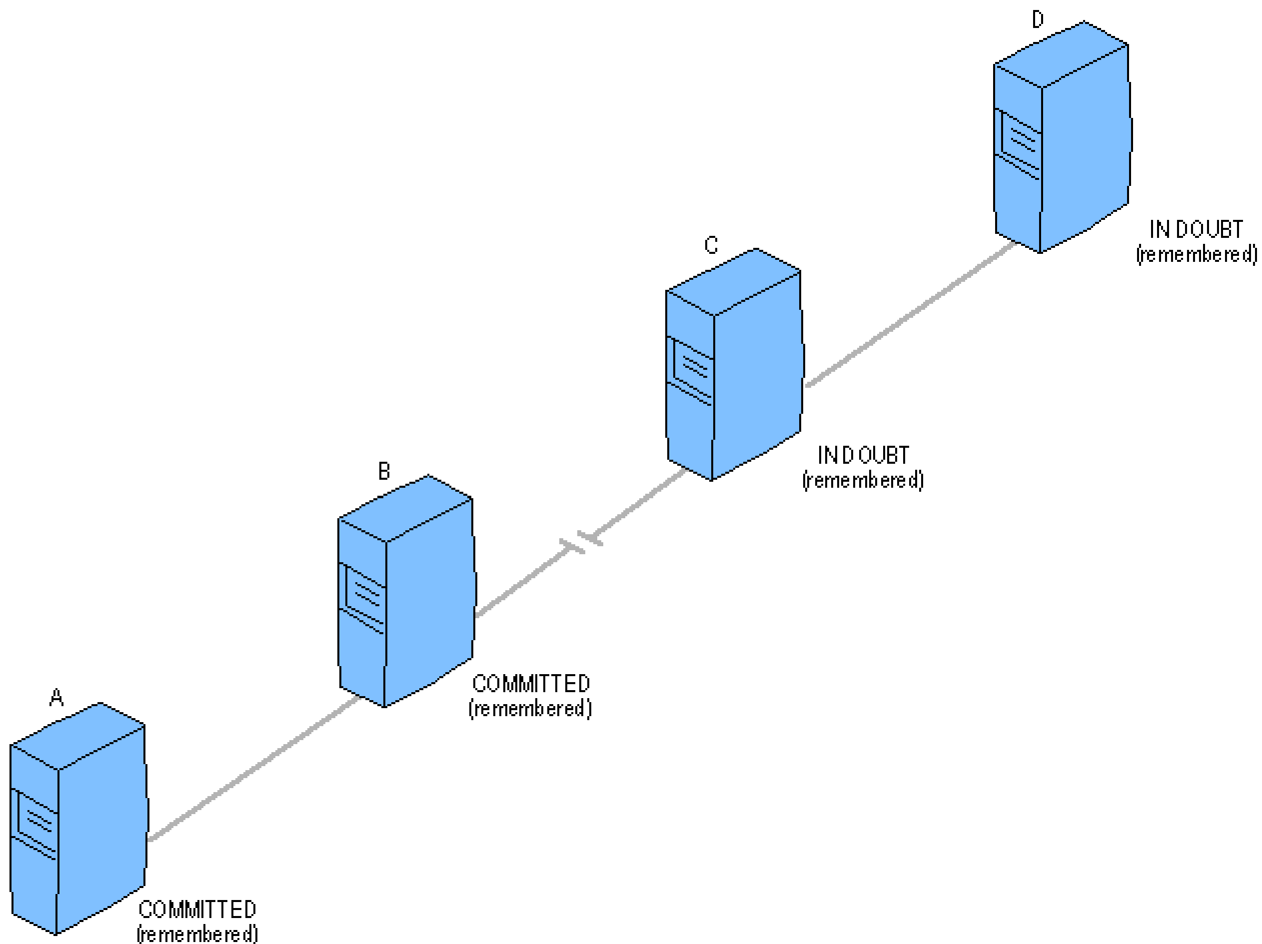
Occasionally, you need to force a transaction either to commit or abort to release locks and make database resources available to other network users and applications.

This can be necessary, for example, when a communication line fails between two computers on the network. Once a transaction has been manually committed or aborted, often it is necessary also to manually force a computer to “forget” the transaction, which deletes the transaction from the local MS DTC log file.

The following illustration shows a case in which a transaction is committed manually. In this example, the following conditions are assumed:

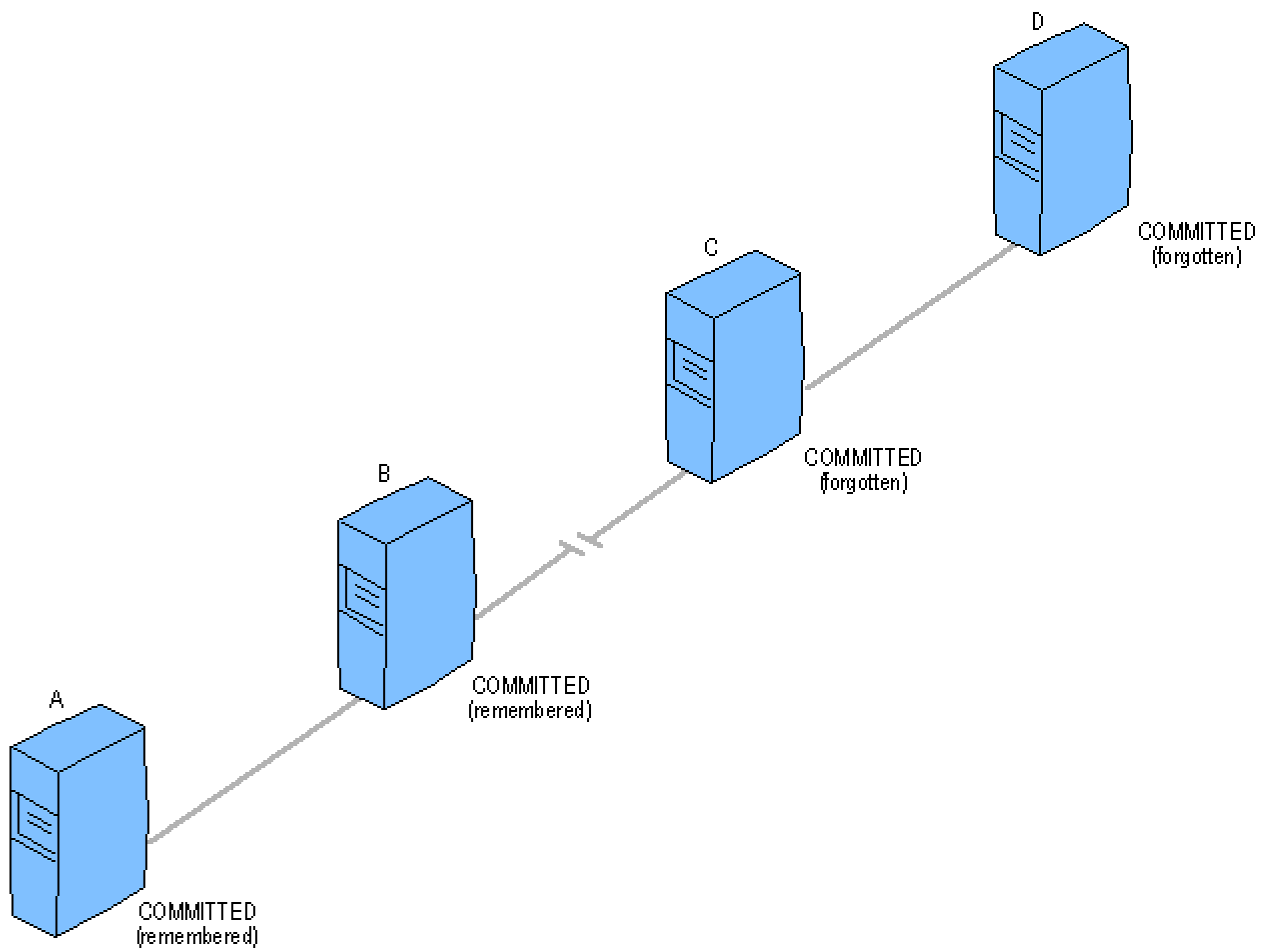
- The MS DTC on computer A is the commit coordinator.
- The lines of communication along which the two-phase commit protocol is conducted proceed sequentially from computer A to computer D.
- The first phase of the two-phase commit protocol has concluded, and MS DTC has written a COMMITTED record to its log.
- Communication fails between computers B and C during the second phase of the two-phase commit protocol.

The transaction is left in the following unresolved state:

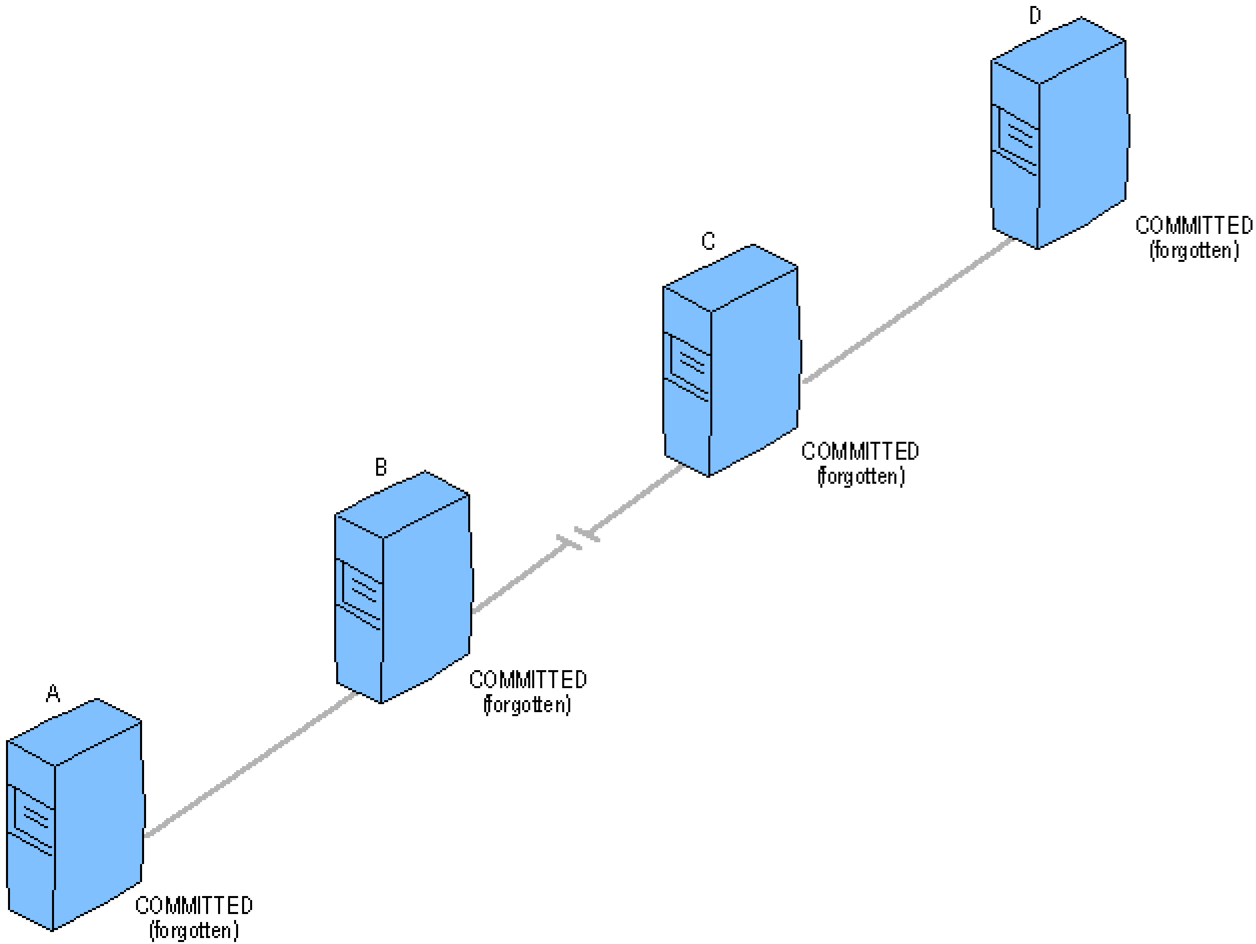


Because the line of communication between computers A and B is still intact, B also has committed the transaction. Both computers, however, must retain the COMMITTED records in their log files until computers C and D confirm that they also have committed the transaction. To resolve the transaction (and thereby release the database locks on computers C and D), the system administrator forces computer C to commit the transaction (see the next illustration).

Because the line of communication between computers C and D is still intact, the forced commit on computer C allows the transaction to commit on computer D. Computer D can now release its database locks and forget the transaction. Once computer D confirms to computer C that it has committed and forgotten the transaction, computer C can also release its locks and forget the transaction.



The transaction is now committed on all computers. However, because computer C cannot communicate its commit to computer B, computer B must continue to remember the transaction. Because computer B has not forgotten the transaction, computer A must also remember it. To complete the transaction, the system administrator forces computer B to forget the transaction (see the next illustration). Computer B's forced forget allows computer A also to forget the transaction. The two-phase commit protocol has been manually concluded, and the transaction is complete.



Important Because of the outgoing-incoming communication pattern of the two-phase commit protocol, it is recommended that you manually resolve transactions on computers immediately adjacent to the break in communications. Therefore, in the preceding example, the forced commit occurs on computer C (not D), and the forced forget occurs on computer B (not A).

Generally, when systems involved in transactions are restarted and connections restored after a system or connection failure, MS DTC will automatically resolve the transactions. MS DTC cannot resolve transactions if the systems are down or connections are not reestablished. In this case, you can manually resolve transactions that are in the In Doubt, Cannot Notify Aborted, or Cannot Notify Committed state if you have a system or connection failure

In Doubt State

The *in-doubt* state indicates that the transaction is on a child, that MS DTC is prepared, and that the parent MS DTC is inaccessible. To resolve the in-doubt transaction, follow these steps:

1. Use the Transaction List window to locate the in-doubt transaction's immediate parent. To do this, right-click the transaction and select the **Properties** command. This displays the parent MS DTC and child MS DTC computers for the transaction.
2. Locate the parent MS DTC and use the Transaction List window on the parent computer to determine the outcome of the in-doubt transaction.
   - If the transaction does not appear in the Transaction List window, then the transaction has been aborted, and you can abort the transaction on the child computer manually.
   - If the transaction appears on the parent computer as Cannot Notify Committed, then the

transaction has committed, and you can commit the transaction on the child computer manually.

- If the transaction appears on the parent computer as Cannot Notify Aborted, then the transaction has aborted, and you can abort the transaction on the child computer manually.
- If the transaction is shown as In Doubt on the parent computer, use the Transaction List window on the parent computer to locate the transaction's next immediate parent. Continue to follow the transaction up the commit tree until you locate the parent on which the transaction is either not shown (indicating that it aborted), in the Cannot Notify Aborted (indicating that it aborted) state, or in the Cannot Notify Committed (indicating that it committed) state. If the transaction is aborted on the parent computer, manually force the transaction to abort on that computer's immediate child. If the transaction is committed on the parent computer, manually force the transaction to commit on the child computer.

3. Once you have either manually committed or aborted the transaction on the child computer, manually force the immediate parent to forget the transaction.

Cannot Notify Committed

The Cannot Notify Committed state indicates that the transaction has committed, but some subordinate MS DTCs have not been notified. You can manually resolve the transaction as follows. Right-click on the transaction that is in the Cannot Notify Committed state. This displays the parent and subordinate MS DTCs for the transaction. Having located the subordinate MS DTCs, force the transaction to commit on each one. Once you have manually committed the transaction on all subordinate MS DTCs, return to the MS DTC that shows the transaction in the Cannot Notify Committed state, and force that MS DTC to forget the transaction.

Caution Do not manually forget a transaction until all subordinate MS DTCs have been notified of the transaction outcome.

Cannot Notify Aborted

The Cannot Notify Aborted state indicates that the transaction has aborted, but some subordinate MS DTCs have not been notified. This state is identical to the Aborting state. You can manually resolve the transaction as follows. Right-click the transaction that is in the Cannot Notify Aborted state. This displays the parent MS DTC and subordinate MS DTCs for the transaction. Having located the subordinate MS DTCs, force the transaction to abort on each one. Once you have manually aborted the transaction on all subordinate MS DTCs, return to the MS DTC that shows the transaction in the Cannot Notify Aborted state, and force that MS DTC to forget the transaction.

Caution Do not manually forget a transaction until all subordinate MS DTCs have been notified of the transaction outcome.

To resolve transactions

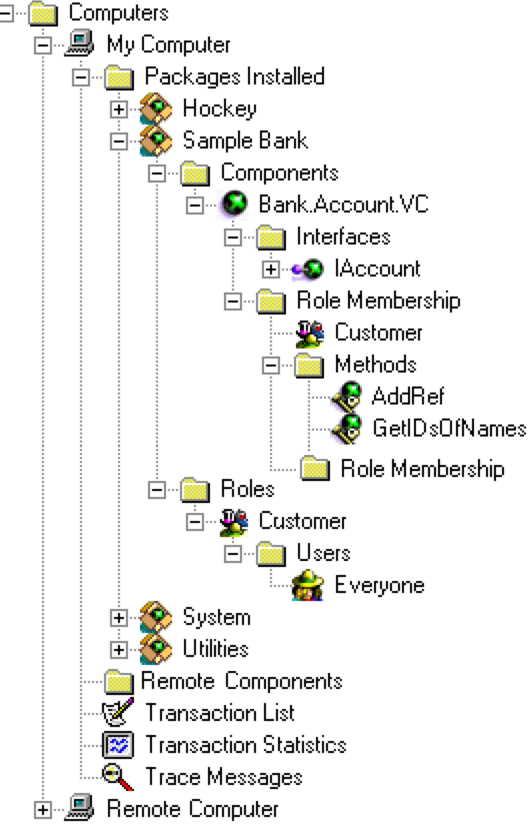
1. In the left pane of the MTS Explorer, select the computer where you want to resolve a transaction.
2. Double-click the Transaction List icon.
3. In the right pane, right-click over the transaction you want to resolve.
4. In the **Resolve** submenu, click **Commit**, **Abort**, or **Forget**.

See Also

Understanding MTS Transactions, Managing MS DTC, Monitoring MTS Transactions, Monitoring MTS Transactions with Windows 95, Understanding MTS Transaction States, Transaction List

Automating MTS Administration

Microsoft Transaction Server (MTS) developers and advanced web administrators can use scriptable administration objects to automate MTS application deployment and administration. The scriptable objects correspond to the Microsoft Transaction Server Explorer collection hierarchy. You can automate administrative procedures by calling interfaces on the appropriate scriptable administration object. The following diagram shows the types of collections administered and deployed by the MTS Explorer:



In order to effectively use the scriptable administration objects, you should have a thorough understanding of the tasks that your application automates in the MTS Explorer. You can use any OLE Automation-compatible language to develop your application because the scriptable administration objects are derived from the **IDispatch** interface. Microsoft Visual Basic™ version 5.0 and Microsoft Visual C++ version 5.0, which support ActiveX technology, are recommended development tools. See the *MTS Administrative Reference* for a complete Visual Basic sample that demonstrates how to use the scriptable administration objects.

This section discusses the following topics:

MTS Administration Objects
Visual Basic Script Sample for Automating MTS Administration
Visual Basic Sample Application for Automating MTS Administration
Automating MTS Administration with Visual Basic
Automating Advanced MTS Administration with Visual Basic

See Also

MTS Administrative Reference

MTS Administration Objects

The following scriptable objects are used for administration:

Catalog Object

CatalogObject Object

CatalogCollection Object

PackageUtil Object

ComponentUtil Object

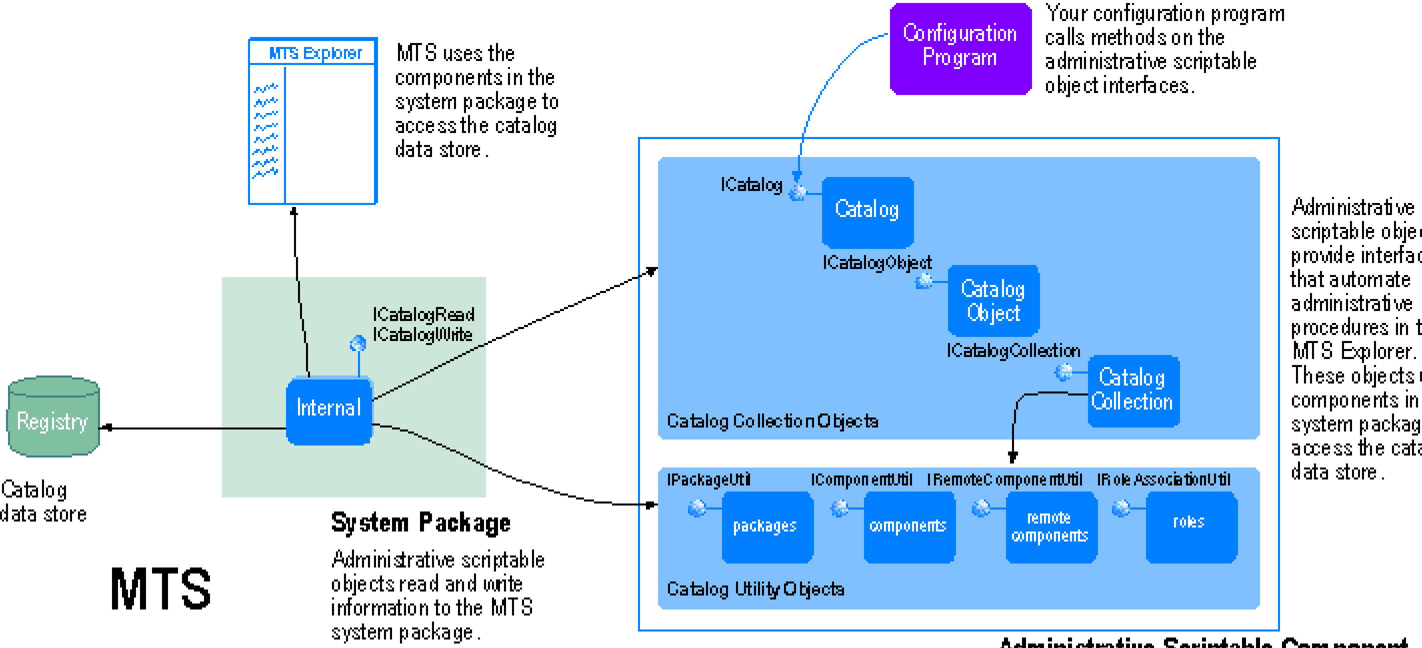
RemoteComponentUtil Object

RoleAssociationUtil Object

These scriptable objects provide general and utility interfaces for your different scripting needs. The Catalog, CatalogObject, and CatalogCollection objects comprise the catalog collection object layer, and 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. The CatalogObject object is used to get and set properties on an object.

In the catalog utility object layer, the PackageUtil, ComponentUtil, RemoteComponentUtil, and RoleAssociationUtil 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 your configuration program uses methods on the administrative objects to read and write data to the catalog.



To learn how to obtain the administrative samples and reference documentation, see the Installing MTS Development Samples and Documentation topic.

See Also

MTS Administrative Reference

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Using MTS Catalog Collection Objects

Each folder in the MTS Explorer corresponds to a collection stored in the catalog. The Catalog, CatalogObject, and CatalogCollection objects comprise the catalog collection object layer. These objects enable you to automate general administrative procedures into your applications such as creating, deleting, or modifying objects.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Instantiating an MTS CatalogCollection Object

You can instantiate a CatalogCollection object in order to access any type of collection or data from any MTS Explorer folder. To instantiate a CatalogCollection object, use the **GetCollection** method and supply a collection name. The CatalogCollection object is a generic object used to access any type of collection (such as data from any Explorer folder). You can call the **GetCollection** method from a Catalog object to get a top-level collection. You can also call the **GetCollection** method from a CatalogCollection object to get a related collection.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Populating an MTS CatalogCollection Object

Use the administrative objects to populate a CatalogCollection object with data from the catalog. Note that instantiating the object does not read any data from the catalog. To browse or change the data in the collection, call the **Populate** or **PopulateByKey** method first. **Populate** will read all the data into the collection. **PopulateByKey** will only read the objects that you specify. You do not need to populate the collection to add data to the collection or use a utility object interface.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Retrieving MTS Objects and Getting/Setting Properties

The CatalogCollection object supports methods for iterating through the objects in the collection. If you are using Visual Basic as your development tool, you can iterate using the **For Each** statement. If you use Visual C++, invoke the **Item** and **Count** methods to enumerate through a collection. Access object properties using the object's **Value** property, and supply a property name as a parameter.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Creating New MTS Objects

Some collections support the **Add** method to create a new object. Other collections require that you use a utility interface to install objects (such as a component). See the Using MTS Catalog Utility Objects topic for more details.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Saving Changes to MTS Objects

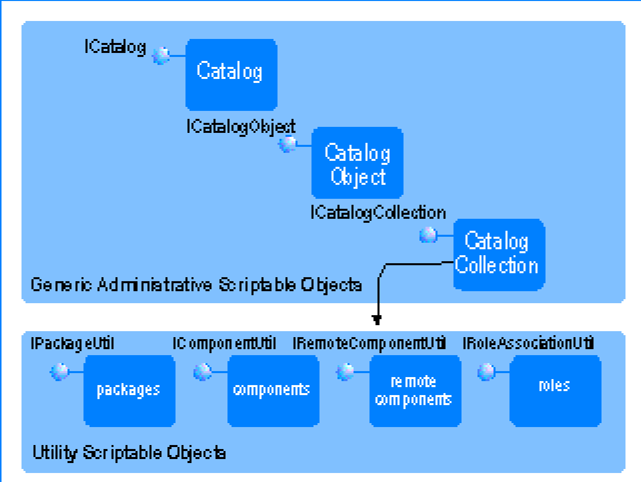
Property changes and new objects (created using the **Add** method) and object removal (deleted using the **Remove** method, which most collections support to delete an object) are held in-memory until you call the **SaveChanges** method. When you call the **SaveChanges** method, any changes in the CatalogCollection object are applied to the catalog. If you release the CatalogCollection object before you call the **SaveChanges** method or call the **Populate** or **PopulateByKe**y method before calling the **SaveChanges** method, all pending changes will be lost.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Using MTS Catalog Utility Objects

The utility object layer consists of the PackageUtil, ComponentUtil, RemoteComponentUtil, and RoleAssociationUtil objects. These objects are used to perform specific actions directly on the catalog (such as installing components or installing pre-built packages). Utility objects are specific to a particular type of collection. Obtain a utility object by calling the **GetUtilInterface** method on the CatalogCollection object.



When using a utility object, the changes made to the catalog do not affect the CatalogCollection object from which you obtained the utility object. You must call the **Populate or PopulateByKey** method on the CatalogCollection object in order to view any changes. There is no need to call the **SaveChanges** method after using a utility method, as changes are written immediately to the catalog.

See Also

MTS Administration Objects, Using MTS Collection Types, MTS Administration Object Methods

Handling MTS Catalog Errors

The catalog collection and catalog utility methods return HRESULTS that indicate success or failure. In Visual Basic, you use the **On Error** handler and the Err object to trap these errors and access the failure code. Methods that deal with many objects (such as the **SaveChanges** or **InstallPackage** methods) may capture multiple error codes to describe specific object failures. You access this set of codes through a collection called **ErrorInfo**. You can use the **GetCollection** method to access an **ErrorInfo** collection. Each CatalogCollection object that you instantiate maintains an **ErrorInfo** collection that stores failure codes for the last method call that failed. If you are installing a package, you can see which components are already installed by accessing the **ErrorInfo** collection for that object.

It is recommended that you program your application to check each method call for success or failure. Your program should especially test for the E_INVALIDARG return code (run-time error 5 in Visual Basic) when supplying collection names or property names. This code indicates that one or more of the supplied collection or property names is not supported.

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, MTS ErrorInfo Collection

Visual Basic Script Samples for Automating MTS Administration

You can use an OLE-Automation compatible language (such as VBScript) to call the scriptable administration objects. MTS provides administrative sample scripts to demonstrate how to use the scriptable objects to automate MTS Explorer procedures.

In order to run an Automation script outside of an HTML page, you must have the Windows Scripting Host (WSH) installed on your computer. WSH is a command-line scripting utility that can be installed by the Window NT 4.0 Option Pack for Windows NT computers. WSH enables scripts to be executed directly on the Windows desktop or command console. You do not need to embed those scripts in an HTML document; instead, scripts can be run directly from the desktop by clicking a script file, or from the command console. For more detailed information, see the Windows Scripting Host documentation.

To demonstrate how to use the administration objects in a script, run the sample scripts contained in the \program files\mtx\samples\WSH sub-directory. The WSH sub-directory contains the following five sample scripts written in VB Script:

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

These scripts automate administrative procedures for Sample Bank. For example, the InstDLL.vbs script calls scriptable objects to first delete existing versions of Sample Bank, create a new package named Sample Bank, install components from the Sample Bank Visual Basic, Visual C++, and Visual J++ DLLs into the new package, change transaction attributes, and add a new role. The InstPak.vbs script also uses the scriptable objects to install Sample Bank, while Uninst.vbs automates uninstalling the Sample Bank package from MTS. After running a script, you can see the results of the script in the MTS Explorer after clicking the **Refresh** toolbar button.

In order to use the InstDLL.vbs, InstPak.vbs, and Uninst.vbs scripts, you must modify the file path in the script to point to the location of the required files on your computer. For example, the file path in the InstPak.vbs script is the default location for the Sample Bank package, which is:

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

If you have the Sample Bank package installed in a difference location, modify that line in the script to point to the current location of the package.

The InstDllCLI.vbs and InstPakCLI.vbs scripts support command line parameters so you can include the file location as a parameter when running your script in a console window. At the command prompt, open the directory containing the script you want to use, and type the name of the script and the location of the package or DLLs. For example, to install the Sample Bank package located in the default directory, open the directory containing the samples scripts and use the following command at the command prompt:

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

To use the InstDLL.vbs script, you must first do one of the following to register the Java components in the vjacct.dll file:

- Install the package by running the InstDLL.vbs script
- Register the Java components using the Microsoft Visual Studio 97 ActiveX Wizard.

See Also

Setting Up the MTS Administrative Sample Scripts

Visual Basic Sample Application for Automating MTS Administration

The Visual Basic version 5.0 sample application demonstrates how to use the methods on the Catalog, CatalogObject, and CatalogCollections objects to automate basic administration functionality for a package named “Scriptable Admin Demo.”

Note You must configure your Visual Basic project to reference the MTS administrative type library (MTSAdmin type library). To reference the MTSAdmin type library, select the **References** option from the Visual Basic Project toolbar. Then browse the available reference files for the “MTS 2.0 Admin Type Library.” For late-binding variables (binding that occurs when you run the program), Visual Basic will locate the type library without further configuration if the MTXADMIN.DLL file is registered on your local machine.

To delete any existing packages named “Scriptable Admin Demo”

1 Call the **CreateObject** method to instantiate a catalog object.

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

2 Get a **Packages** collection object by calling the **GetCollection** method. The **Packages** collection returns without retrieving any data from the catalog so that the collection will be empty upon return from the **GetCollection** method.

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

3 Find the previous version of the "Scriptable Admin Demo" package by populating the **Packages** collection to read in all packages and search for "Scriptable Admin Demo." Enumerate through the collection, starting at the highest index so the **Remove** method can be called from within the loop. The **Remove** method releases the object, removes the object from the collection, and shifts the objects in the collection so that object(*n+1*) becomes object(*n*) for all *n* greater than or equal to the index being removed. The effect of **Remove** method on the collection object is immediate. The **Item** and **Count** methods called any time after the **Remove** method will reflect the change in the index. However, the removal of the package is not applied to the catalog until the **SaveChanges** method is called (see step 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 Call the **SaveChanges** method to save changes to the data store.

```
packages.SaveChanges
```

To create a new package named “Scriptable Admin Demo Package”

1 Add a new package using the **Add** method, and note the package ID assigned. The **Add** method adds the object to the collection but does not apply the changes to the catalog until the **SaveChanges** method is called (see step 3). Note that the **Add** method will apply default values to all properties. The default ID will be a new unique identifier.

```
Dim newPack As Object
Dim newPackID As Variant
Set newPack = packages.Add
newPackID = newPack.Value("ID")
```

2 Update the **Name** and **SecurityEnabled** properties.

```
newPack.Value("Name") = "Scriptable Admin Demo"
newPack.Value("SecurityEnabled") = "N"
```

3 Call the **SaveChanges** method to save the new package to the catalog. The return value of this call is the number of objects changed, added, or deleted. If no changes were pending, the method returns 0.

```
n = packages.SaveChanges
```

To update the “Scriptable Admin Demo” package properties and get the ComponentsInPackage collection.

1 Call the **PopulateByKey** method to read the package back from the catalog. Pass an array containing the keys of the objects to read. In the sample code, we use an array containing a single element (the ID of the package just created).

```
Dim keys(0) as Variant
keys(0) = newPackId
packages.PopulateByKey keys
```

2 Get the package object from the collection

```
Dim package As Object
Set package = packages.Item(0)
```

3 Update the **SecurityEnabled** property for the package.

```
package.Value("SecurityEnabled") = "Y"
```

4 Call the **GetCollection** method to retrieve the ComponentsInPackage collection. Supply the key of the "Scriptable Admin Demo package as a parameter.

```
Set components = packages.GetCollection("ComponentsInPackage",_
package.Key)
```

5 Call the **SaveChanges** method to save the changes to the catalog.

```
packages.SaveChanges
```

To install a componentasdefcomponent into the "Scriptable Admin Demo package":

1 Call the **GetUtilInterface** method to get the component utility object. This object is used to install components.

```
Dim util As Object
Set util = components.GetUtilInterface
On Error GoTo installFailed
```

2 Call the **InstallComponent** method, passing in a string containing the name of the dynamic-link library (DLL) of the component to be installed. If the component does not have an external type library or a *proxy-stub DLL*, pass in empty strings as the second and third arguments. Note that you do not have to call the **SaveChanges** method after installing a new component. All components contained in a DLL will be installed by this method, and are immediately written to the catalog. Call the GetCLSIDs method to get the CLSIDs of the components installed.

```
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 Call the **PopulateByKey** method to read back the components just installed. Note that the components installed into the package via the **InstallComponent** method are not visible in the collection until the **Populate or PopulateByKey** method is called to read the data from the catalog.

```
components.PopulateByKey installedCLSIDs
```

To find and delete the Bank.CreateTable component from the "Scriptable Admin Demo package":

1 Iterate through the components and change transaction attributes using the **Item** and **Count** methods.

```
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 Find and delete the `Bank.CreateTable` component by index. Note that you must iterate though the collection backwards in order to call the **Remove** method during the loop.

```
If component.Value("ProgID") = "Bank.CreateTable" Then
        components.Remove (i)
    End If
Next
```

3 Retrieve a new count and iterate through the collection again. Note that the `Bank.CreateTable` component will not be deleted from the data store until the **SaveChanges** method is called. Display a message box that informs the user if the installation succeeded.

```
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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic, Automating Advanced MTS Administration with Visual Basic

Automating MTS Administration with Visual Basic

The scriptable administration objects can be used to install, delete, and update properties for packages and components. These topics provide a sample procedure and Visual Basic version 5.0 sample code using methods on the scriptable administration objects to automate the following administrative tasks:

- Automating Installation of a Pre-Built MTS Package
- Automating a New MTS Package and Installing Components
- Automating Enumerating Through Installed MTS Packages to Update Properties
- Automating Enumerating Through Installed MTS Packages to Delete a Package
- Automating Enumerating Through Installed MTS Components to Delete a Component

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Installation of a Pre-Built MTS Package

To install a pre-built package named "Test.pak" into the MTS Explorer:

1 Declare the objects that you will be using to install a pre-built package.

```
Private Sub InstallPackage_Click()
 Dim catalog As Object
 Dim packages 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 instantiate the Catalog object. Retrieve the top level Packages collection by calling the **GetCollection** method.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
```

4 Instantiate the PackageUtil object, and call the **InstallPackage** method to install a package named "test.pak."

```
Set util = packages.GetUtilInterface
 util.InstallPackage "c:\test.pak", "", 0
 Exit Sub
```

5 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Creating a New MTS Package and Installing Components

To create a new package named “My Package” and install the components in that package:

1. Declare the objects that you will be using to create a new package and install components into that 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 instantiate 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Enumerating Through Installed MTS Packages to Update Properties

To enumerate through installed packages in order to update properties in the package named “My Package”:

1. Declare the objects that you will be using to enumerate through installed packages to update package properties.

```
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. 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 instantiate the Catalog object. Retrieve the Packages collection by calling the **GetCollection** method. Then call the **Populate** method to fill the collection with packages from the catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Enumerate through a collection to find the package named “My Package.” When “My Package” is located, set the **SecurityEnabled** property to “Y.” Call the **SaveChanges** method to save the property update for the package.

```
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. 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Enumerating Through Installed MTS Packages to Delete a Package

▸To enumerate through installed packages to delete the package named "My Package":

1. Declare the objects that you will be using to enumerate through installed packages to delete a specific package.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim componentsInPack 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 instantiate the Catalog object. Retrieve the **Packages** collection by calling the **GetCollection** method. Then call the **Populate** method to fill the collection with packages installed in the catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Use the **Count** and **Item** methods to enumerate through the package collection to find the package named "My Package." When "My Package" is located, call the **Remove** method to delete the package. Then save changes to the collection by calling the **SaveChanges** method.

```
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. 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Enumerating Through Installed MTS Components to Delete a Component

To enumerate through installed components to delete a component:

1. Declare the objects that you will be using to enumerate through installed components to delete a specific component.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim componentsInPack 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 instantiate the Catalog object. Retrieve the Packages collection by calling the **GetCollection** method. Then call the **Populate** method to fill the collection with packages installed in the catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Enumerate through the collection to look for the package named “My Package.” Then call the **GetCollection** method to get the ComponentsInPackage collection. Fill the ComponentInPackages collection using the **Populate** method, and then enumerate through the collection to find the “Bank.Account” component. Call the **Remove** method to delete the component, and save changes to the collection by calling the **SaveChanges** method.

```
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. 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating Advanced MTS Administration with Visual Basic

Automating Advanced MTS Administration with Visual Basic

The scriptable administration objects can be used to configure clients and roles, export packages, and access the names of collections and properties supported by the catalog. The following topics provide a procedure and Visual Basic version 5.0 sample code using methods on the scriptable administration objects to automate the following administrative tasks:

- Automating Access to MTS Related Collection Names
- Automating Access to MTS Property Information
- Automating MTS Role Configuration
- Automating MTS Package Export
- Automating Configuration of an MTS Client to Use Remote Components
- Automating MTS Package Property Updates on Remote Servers

Automating Access to MTS Related Collection Names

The RelatedCollectionInfo collection provides a list of related collections that you can access from a given collection. See the RelatedCollectionInfo topic in the MTS Administrative Reference for a list of properties supported by this collection.

To access and display related collection names:

1. Declare the objects that you will be using to access the object that provides the names of related collections

```
Dim catalog As Object
Dim packages As Object
Dim RelatedCollectionInfo As Object
Dim collName 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 instantiate the Catalog object. Retrieve the Packages collection by calling the **GetCollection** method. Then call the **GetCollection** method on the Packages collection object to obtain the RelatedCollectionInfo collection. Note that the key value is left blank when calling **GetCollection** to access the RelatedCollectionInfo collection. The key value is not used because the information in RelatedCollectionInfo will be the same for all packages. Fill the RelatedCollectionInfo collection with information from the catalog by calling the **Populate** method.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set RelatedCollectionInfo =_
packages.GetCollection("RelatedCollectionInfo", "")
RelatedCollectionInfo.Populate
```

4. Enumerate through the RelatedCollectionInfo collection and display the names of each collection.

```
For Each collName In RelatedCollectionInfo
   Debug.Print collName.Name
Next

Exit Sub
```

5. 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic

Automating Access to MTS Property Information

The PropertyInfo collection stores information about each property in a collection. See the PropertyInfo topic in the MTS Administrative Reference for more information about this collection.

To access and list the name of each property in a collection:

1. Declare the objects that you will be using to access property information stored in the catalog.

```
Dim catalog As Object
Dim packages As Object
Dim propertyInfo As Object
Dim property 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 instantiate the Catalog object. Retrieve the **Packages** collection by calling the **GetCollection** method. Call the **GetCollection** method on the **Packages** collection to obtain the PropertyInfo collection. Note that the key value is left blank when calling **GetCollection** to access the PropertyInfo collection. The key value is not used because the information in PropertyInfo will be the same for all packages. Fill the PropertyInfo collection with information from the Catalog by using the **Populate** method.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
Set propertyInfo = packages.GetCollection("PropertyInfo", "")
propertyInfo.Populate
```

4. Enumerate through the PropertyInfo collection and list the names of each property in the collection.

```
For Each property In propertyInfo
        Debug.Print property.Name
    Next

    Exit Sub
```

5. 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
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic

Automating MTS Role Configuration

To configure a role a package and component, and assign administrative privileges to a user:

1. Declare the objects that you will be using to configure a role for a specific component.

```
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. 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 instantiate the Catalog object. Retrieve the Packages collection by calling the **GetCollection** method. Then populate the Packages collections with data from the catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Enumerate through the Packages collection to look for the package named “My Package.” When “My Package” is located, call the **GetCollection** method to obtain the RolesInPackage collection. Add a new role to package using the **Add** method. Name the new role “R1,” and save changes to the collection.

```
    If pack.Name = "My Package" Then
  Set rolesInPack = packages.GetCollection("RolesInPackage", pack.Key)
            Set newRole = rolesInPack.Add
            newRole.Value("Name") = "R1"
            rolesInPack.savechanges
```

5. Call the **GetCollection** method on the RolesInPackage collection to get the UsersInRole collection. Use the **Add** function to add an existing NT user to the role. Set the user name to “administrator.” Save changes to the UsersInRole collection.

```
            Set usersInRole = RolesInPackage.GetCollection("UsersInRole",
            newRole.Key)
            Set newUser = usersInRole.Add
            newUser.Value("User") = "administrator"
            usersInRole.savechanges
```

6. Get the ComponentsInPackage collection using the **GetCollection** method. Populate the ComponentsInPackage collection, and enumerate through the collection to find the Bank.Account component. To associate the new role with the component, instantiate the RoleAssociationUtil object using the **GetUtilInterface** method. Then associate the new role with the component by calling the **AssociateRole** method.

```
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 Use the Err object to display an error message if the installation of the component fails.

```
failed:
 MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic

Automating MTS Package Export

To export a package named “test.pak”:

1. Declare the objects that you will be using to export a package.

```
Dim catalog As Object
Dim packages 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 instantiate the Catalog object. Retrieve the Packages collection by calling the **GetCollection** method. Call the **Populate** method to fill the package with data from the catalog.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set packages = catalog.GetCollection("Packages")
packages.Populate
```

4. Enumerate through the Packages collection to find the package named “My Package.” Once that package is located, instantiate the package utility object and call the **ExportPackage** method.

```
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. Use the Err object to display an error message if the installation of the component fails.

```
failed:
    MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic

Automating Configuration of an MTS Client to Use Remote Components

▶ **To configure a client to use the Bank.CreateTable remote component:**

1 Declare the objects that you will be using to configure a client (running MTS) to run remote components.

```
Dim catalog As Object
Dim remoteComps 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 instantiate the Catalog object. Retrieve the RemoteComponents collection by calling the **GetCollection** method. Then instantiate the RemoteComponentUtil object by using the **GetUtilInterface** method. To install the remote component, call the **InstallRemoteComponentByName** method and supply the name of the server computer, the package name on the server, and the component name.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set remoteComps = catalog.GetCollection("RemoteComponents")
Set util = remoteComps.GetUtilInterface
   util.InstallRemoteComponentByName "remote1", "New","Bank.CreateTable"
Exit Sub
```

4 Use the **Err** object to display an error message if the installation of the component fails.

```
failed:
 MsgBox "Failure code " + Str$(Err.Number)

End Sub
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with Visual Basic

Automating MTS Package Property Updates on Remote Servers

To update package properties on a remote computer named "remote1":

1. Declare the objects that you will be using to configure a client (running MTS) to deploy and administrator remote components.

```
Dim catalog As Object
Dim packages As Object
Dim pack As Object
Dim root 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 instantiate the Catalog object. Call the **Connect** method to access the ***root collection*** on the computer named "remote1." The root collection is a collection object that can be used to access top-level collections on the given computer. The root collection contains no objects and has no properties. Note that the key value is not used when calling **GetCollection** from a root collection. Get the Packages collection on the remote computer by calling the **GetCollection** method. Then use the **Populate** method to fill the packages collection.

```
Set catalog = CreateObject("MTSAdmin.Catalog.1")
Set root = catalog.Connect("remote1")
Set packages = root.GetCollection("Packages", "")
packages.Populate
```

4. Set the SecurityEnabled setting to "Y" for "My Package" and save changes to the package collection.

```
 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. Use the Err object to display an error message if the installation of the component fails.

```
   failed:
    MsgBox "Failure code " + Str$(Err.Number)

   End Sub
```

See Also

MTS Administration Objects, MTS Collection Types, MTS Administration Object Methods, Automating MTS Administration with 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.

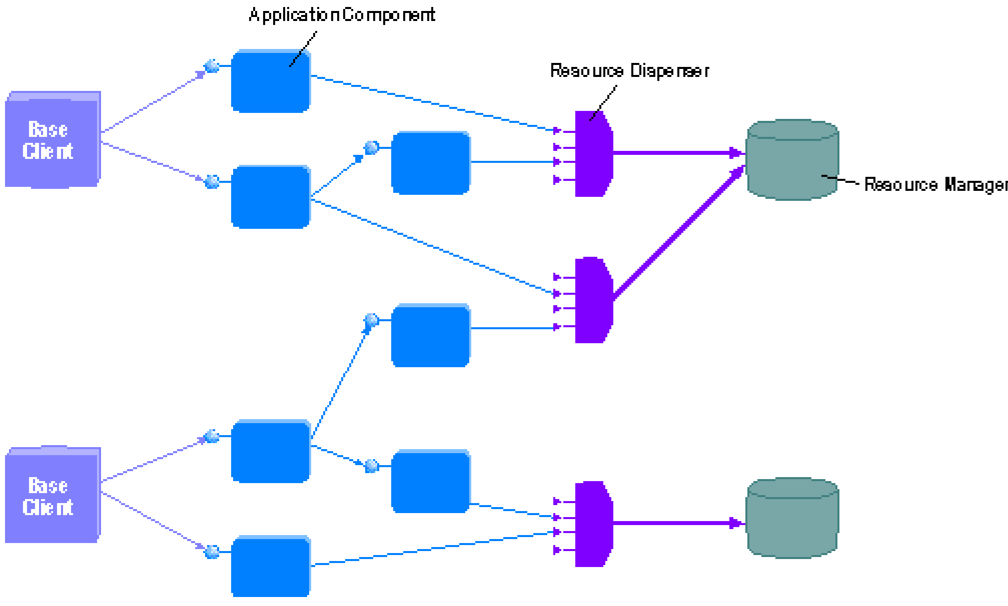
MTS Programming Concepts

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

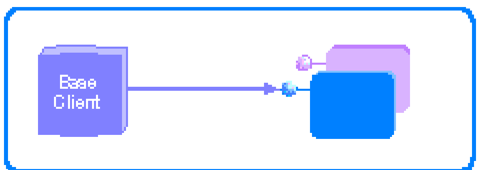
Application Components

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.

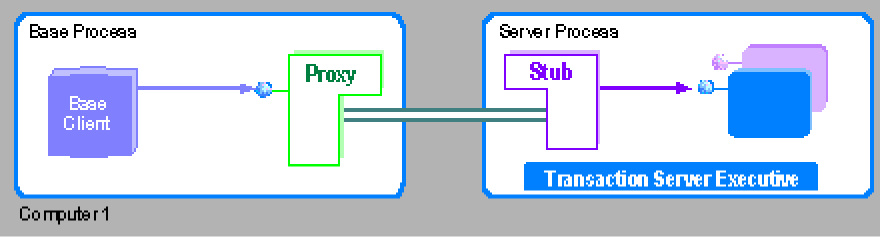
MTS shelters developers from complex server issues, allowing them to focus on implementing business functions. Because components running in the MTS run-time environment can take advantage of transactions, developers can write applications as if they run in isolation. MTS handles the concurrency, resource 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).

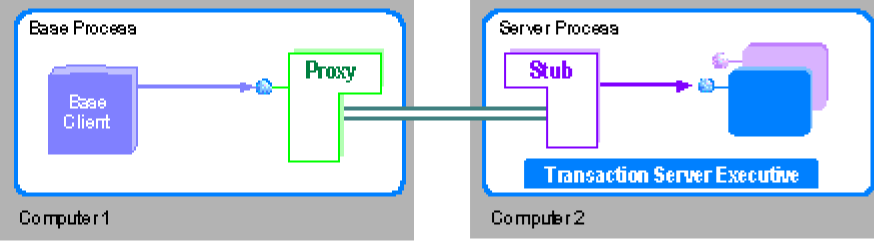
In-process, local, and remote components



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

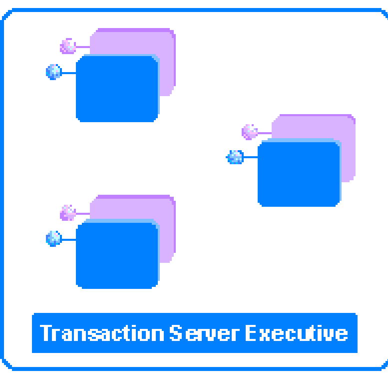
Application Components

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.

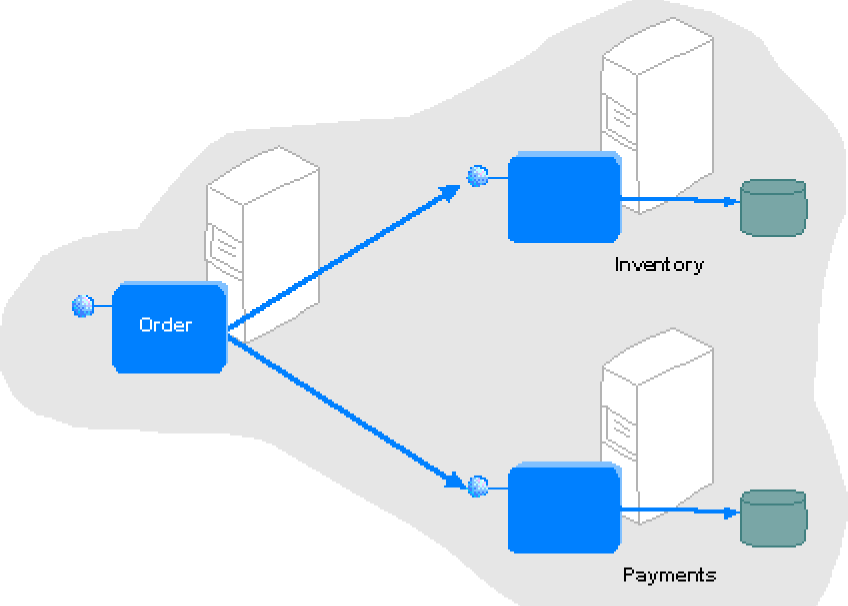
Transactions maintain the ACID properties:

- *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.

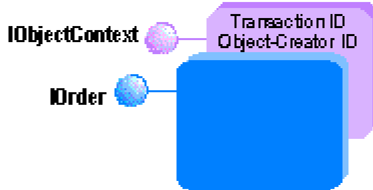
On clustered servers, MTS will not automatically reconnect to MS DTC in the event of a failover. Not enough information exists about the transaction composition and state to determine the appropriate course of action. Retries remain the responsibility of the client application. The client cannot differentiate 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.

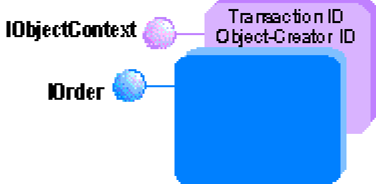
See Also

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**.

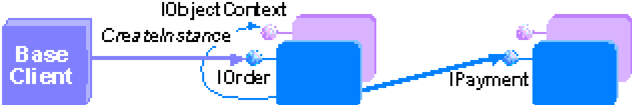
Note If you are using Visual C++ and running an MTS component in-process, you must:

- 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

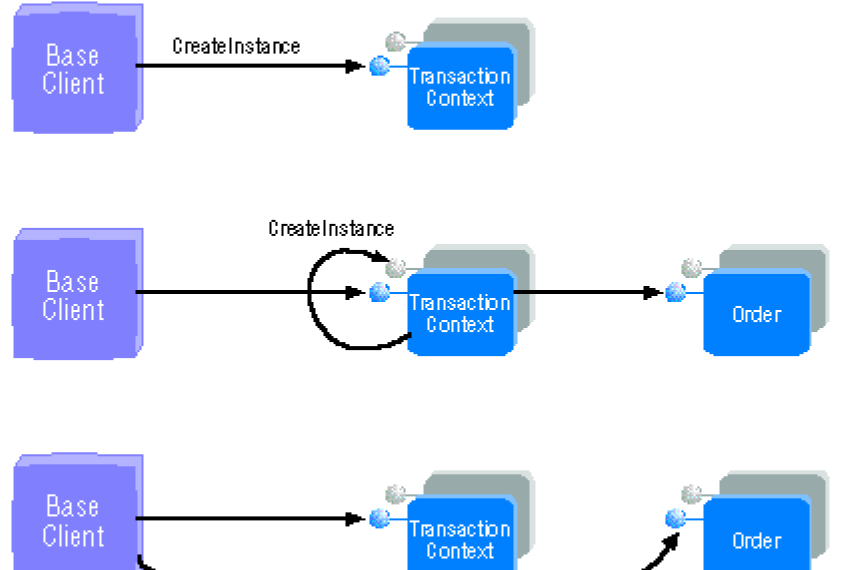
MTS Objects, Calling MTS Components, Context Objects, Passing Object References, Deactivating Objects, **CreateInstance**

Transaction Context Objects

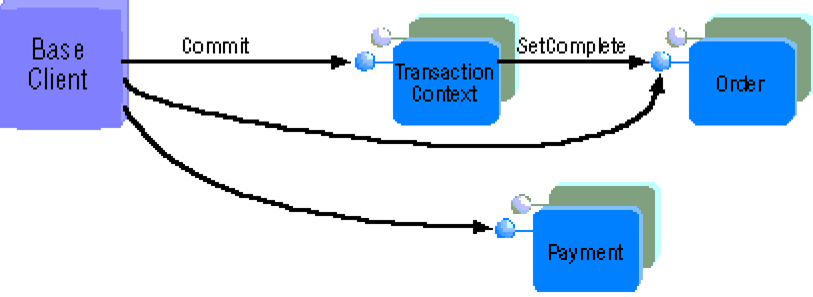
The *transaction context object* allows base clients to combine the work of multiple MTS objects into a single transaction, without having to develop a new component specifically for that purpose.

The transaction context object's methods use its context object as follows:

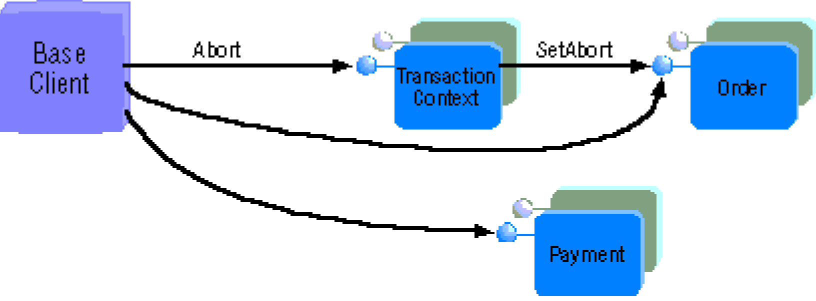
- **CreateInstance** – Calls **CreateInstance** and returns a reference to the newly created object.



- **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:

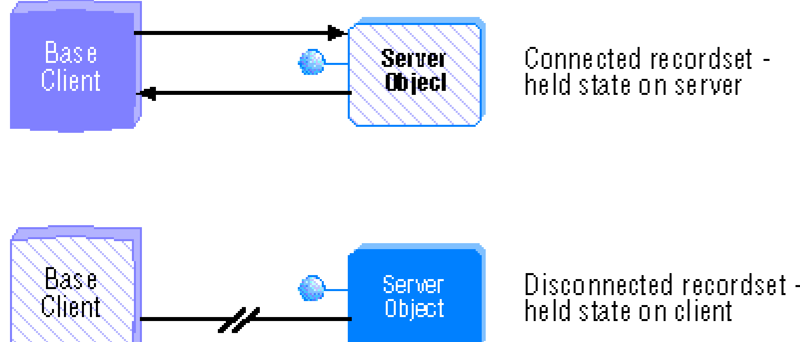
Pass Parameter	If	Client Requirements
By value	Data is relatively small	Recipient requires all data and can get data without reaccessing caller.
By reference	Data is relatively large	Recipient does not require all data and must reaccess caller, possibly many times.

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.

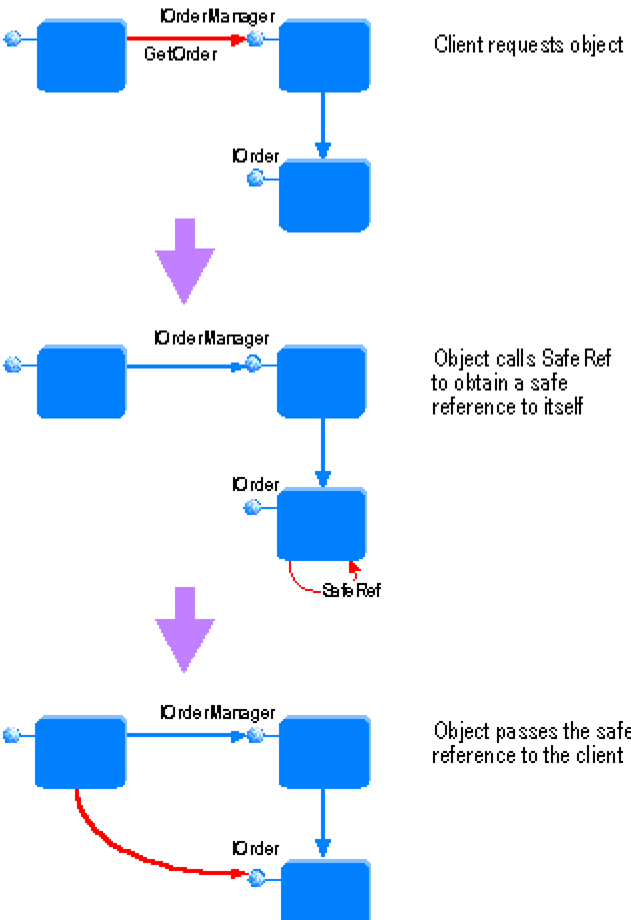
Passing Object References

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

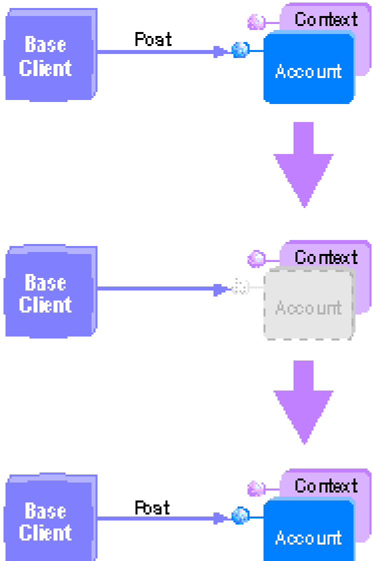
MTS extends COM to allow object deactivation, even while client references are maintained. This makes server applications more scalable by allowing server resources to be used more efficiently.

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

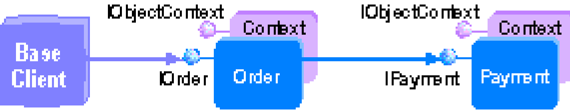
Deactivating Objects, **IObjectControl**

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.

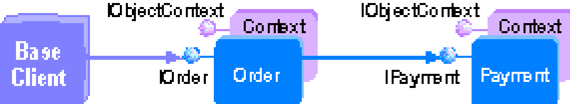
Roles are defined during application development. These roles are then assigned to specific users at deployment time.

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.

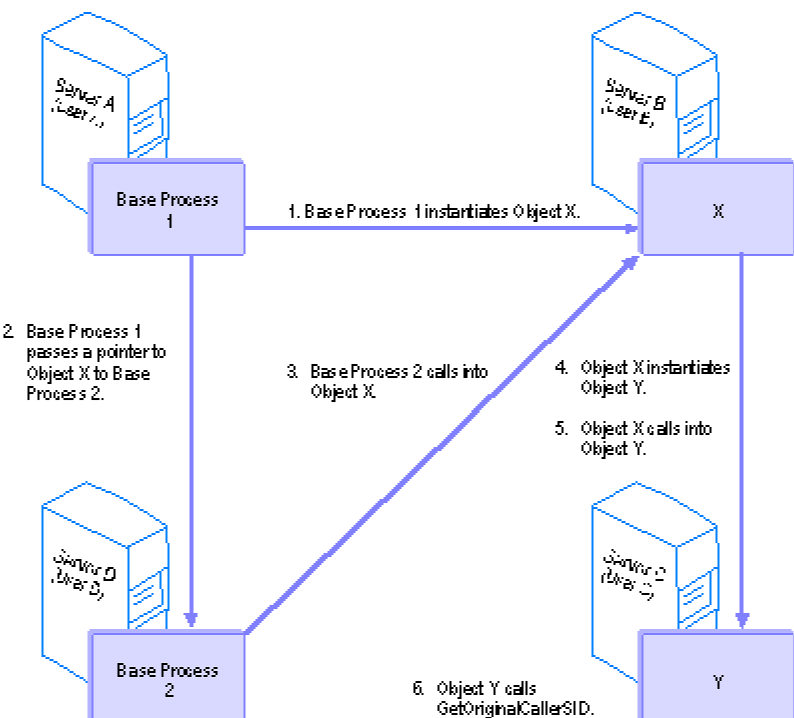
Note Visual Basic programmers can use the **SecurityProperty** object. The methods for **SecurityProperty** return user name strings instead of security identifiers (SIDs).

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.

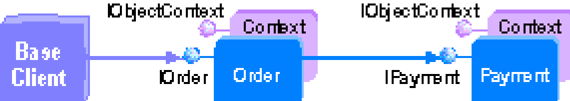


Calling the following methods on Object Y returns SIDs associated with these users:

- **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)

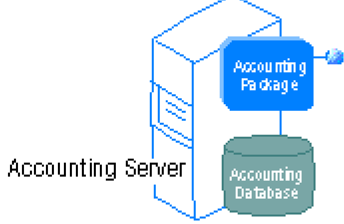
MTS provides a way to set up remote components by using the Remote Computer and Remote Component folders in the MTS Explorer hierarchy. For more information about "pulling" or "pushing" components between computers, see the *Administrator's Guide*.

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

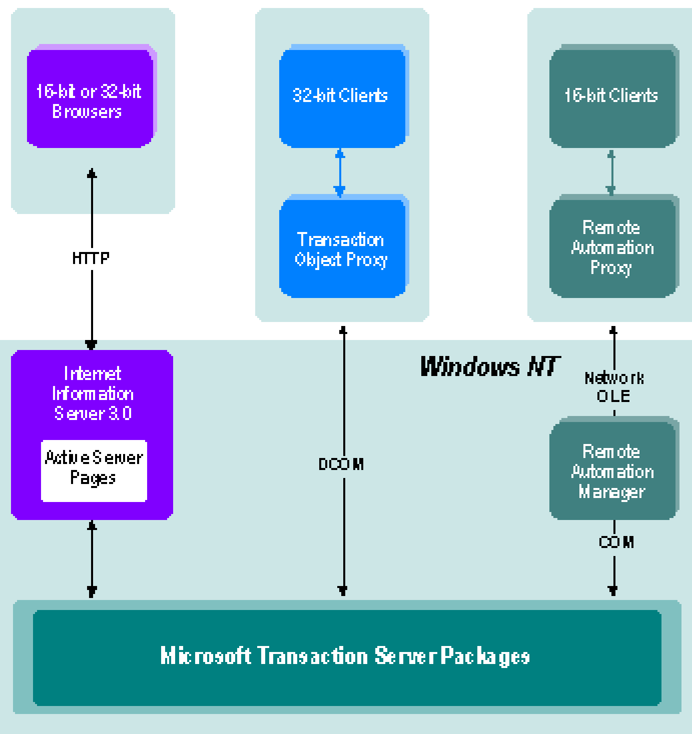
This topic covers the following:

- 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.

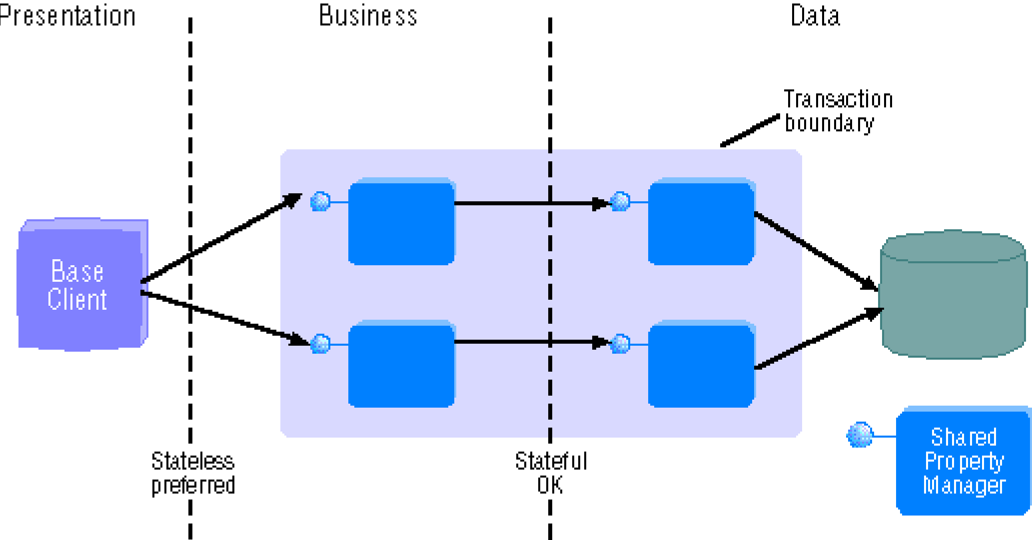
You must configure the following registry key to the Prog ID of the object that you want to call:

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.

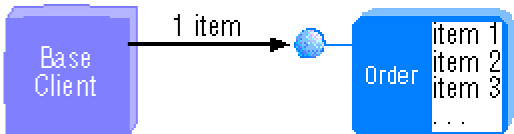
Example: Order-Entry Application

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:

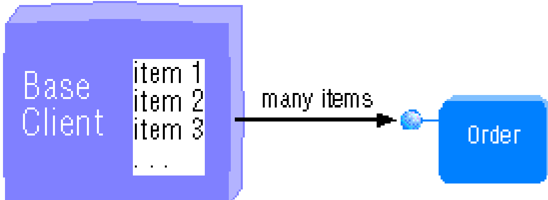


This application has the following properties:

- 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*

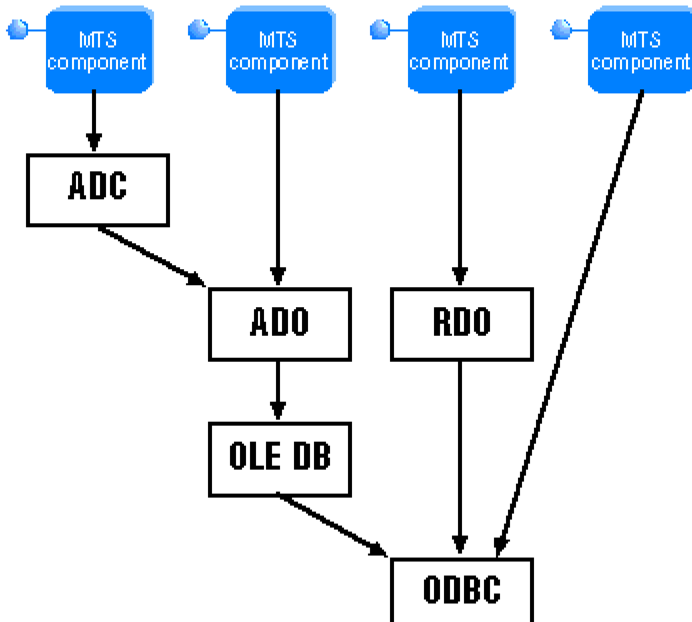
that ensures that the transaction outcome (either *commit* or *abort*) is consistent across all resource managers involved in a transaction. DTC supports resource managers that implement OLE Transactions, X/Open XA protocols, and LU 6.2 Sync Level 2.

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*.

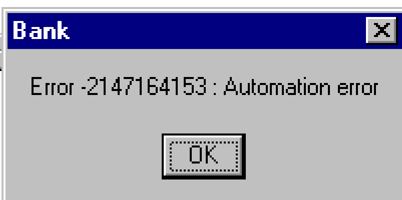
MTS Error Diagnosis

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
6/12/97	1:27:48 PM	LicenseService	None	213
6/12/97	1:12:47 PM	LicenseService	None	213
6/12/97	12:57:47 PM	LicenseService	None	213
6/12/97	12:42:42 PM	LicenseService	None	213

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:

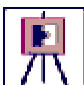
- Make sure that the Distributed Transaction Coordinator (DTC) is running on all servers.
- Check network communication by first testing on a local computer to verify that the application works. If you are running TCP/IP on your network, you can then use the Windows NT Ping.exe utility to verify that 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.

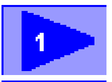
Creating a Simple ActiveX Component

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).



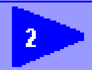
Scenario: Creating and Using a Simple ActiveX Component

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.



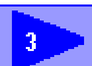
Creating the Account Component

Create a new ActiveX component, Account.



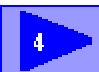
Creating the Bank Package

Use the MTS Explorer to create a new package for your component.



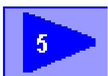
Installing the Account Component in the Bank Package

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

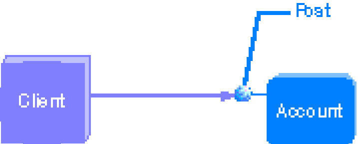
Learn how to efficiently share resources, such as database connections, through the MTS Resource Dispenser Manager and resource dispensers.

See Also

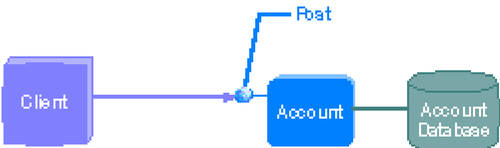
Programming Concepts, Transaction Server Components, Building Scalable Components

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).

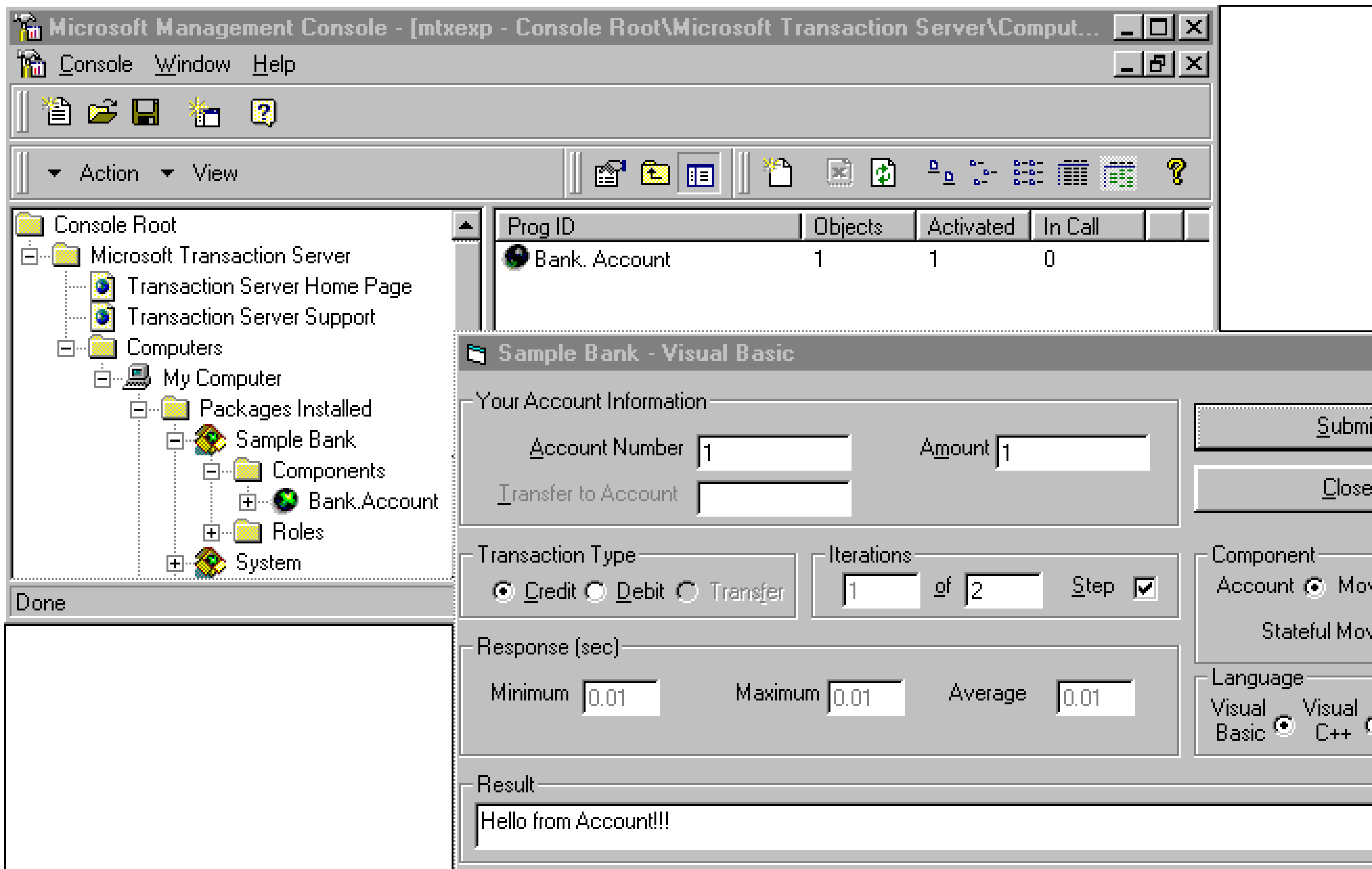
To run and monitor your component

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.

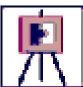
The topic Building Scalable Components, shows you how to use just-in-time activation to use server resources even more efficiently, resulting in more scalable applications and improved performance.

See Also

Application Design Notes: Resource Usage, Building Scalable Components, Modifying the Account Component, Creating the Account Component, Creating a Simple ActiveX Component

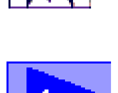
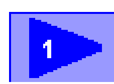
Building Scalable Components

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.



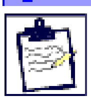
Scenario: Adding Just-In-Time Activation to the Account Component

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

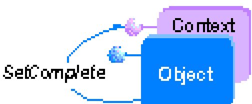
In this scenario, the Bank client creates an Account object from the Account component and calls its Post method, just as in Creating a Simple ActiveX Component. This time, Account obtains a reference to its context object. When it successfully completes its work on behalf of the client, it uses its 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

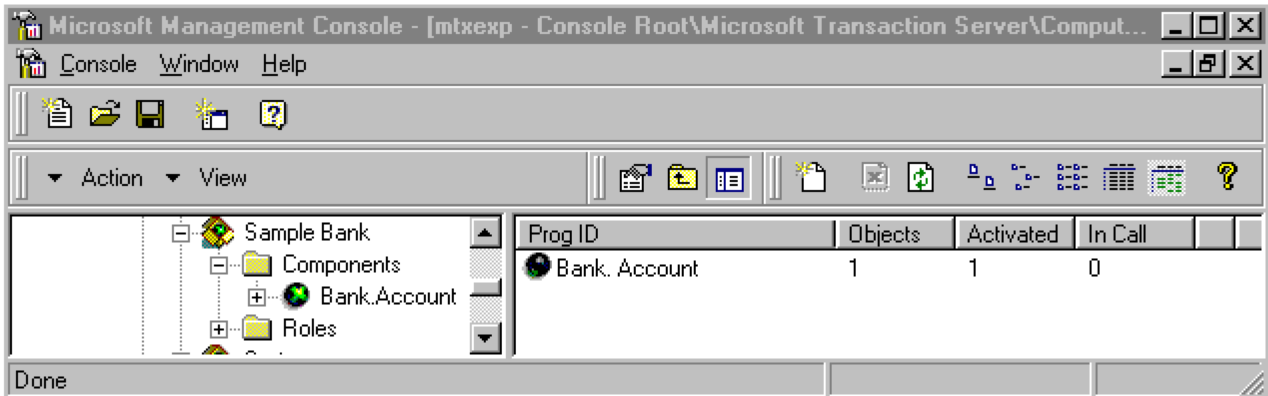
2 Build the component as a DLL and save it as \Mts\Samples\Account.VB\Step3\VBAcct.dll.

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.

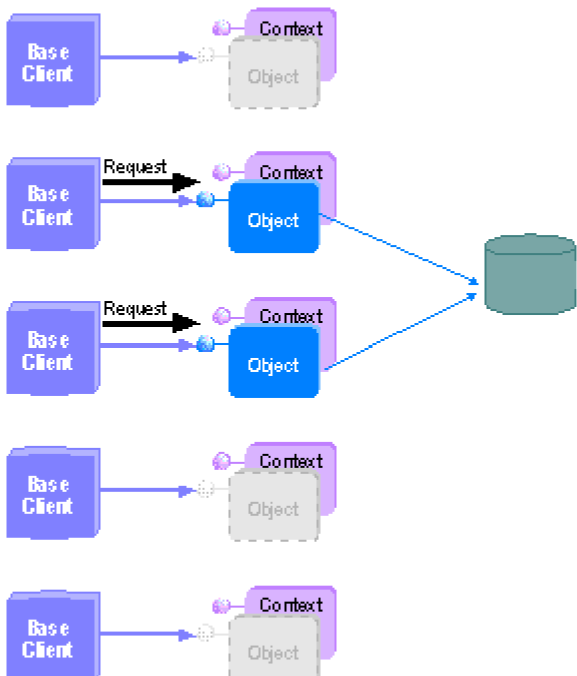
  The advantage to this technique is that it conserves server resources. The disadvantage is that, as your application scales up, your performance slows down. If the object is on a remote computer, each time an object is created, there must be a network round-trip, which negatively affects performance.
- 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

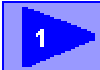
Context Objects, Deactivating Objects, Creating a Simple ActiveX Component, **GetObjectContext** method, **SetAbort** method, **SetComplete** method

Building Transactional Components

This section introduces transactional components and the benefits of running components within the same transaction.

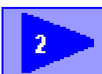
Scenario: Composing Work from Multiple Components Under the Same Transaction

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

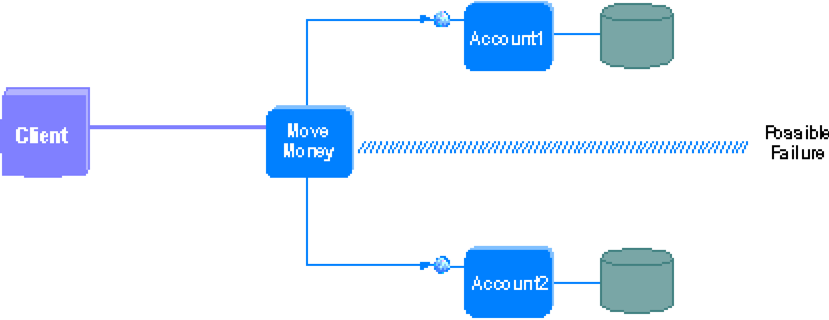
Using transactional components provides atomicity and simplified error recovery.

See Also

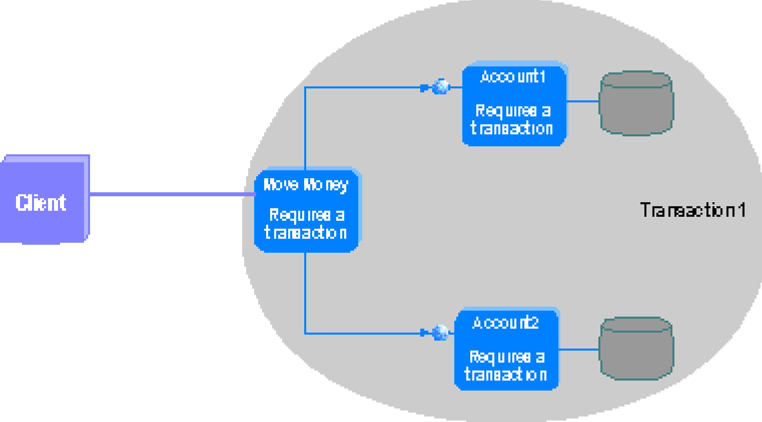
Transactions, Transaction Attributes, **ObjectContext** object, **CreateInstance** method

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.

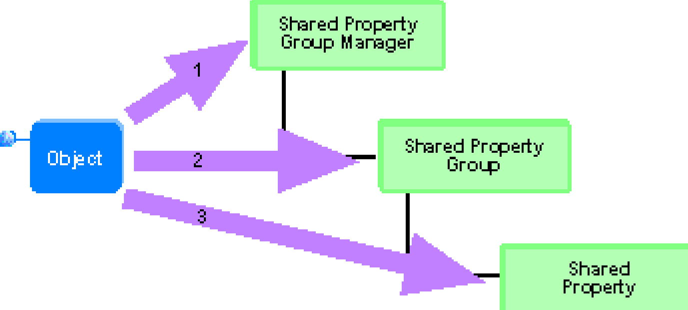
See Also

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.

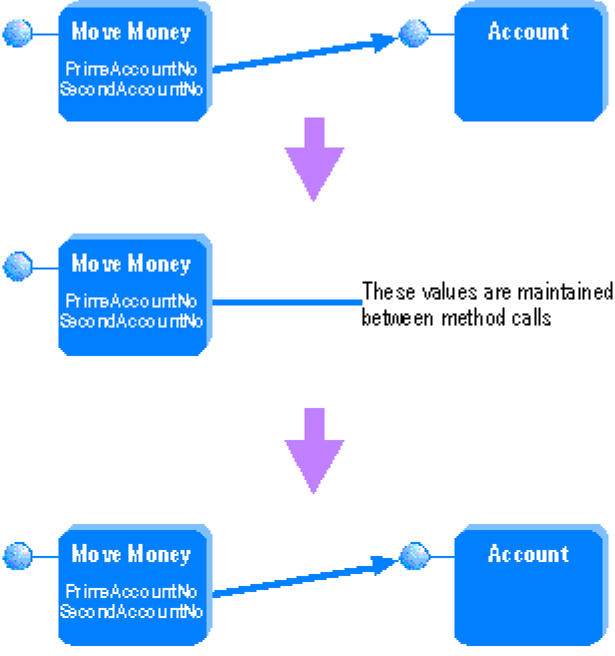
See Also

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.

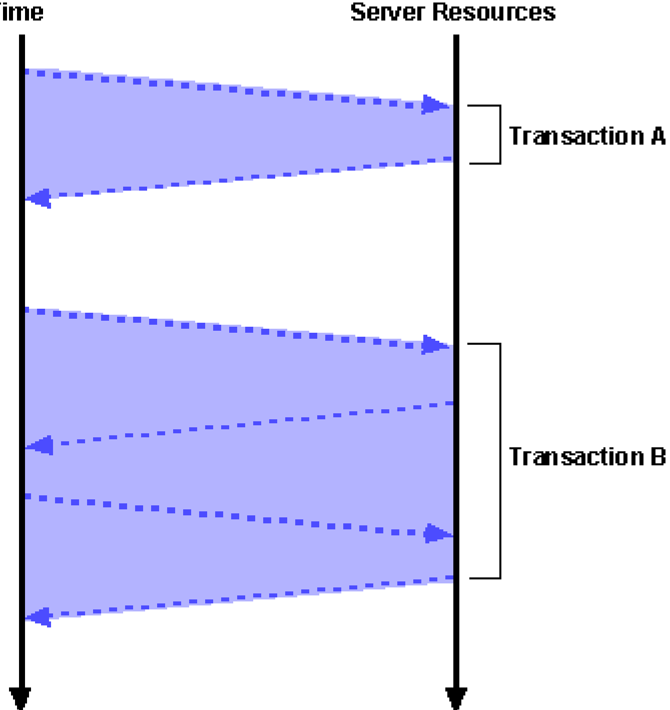
Why does MoveMoney outperform Stateful MoveMoney?

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

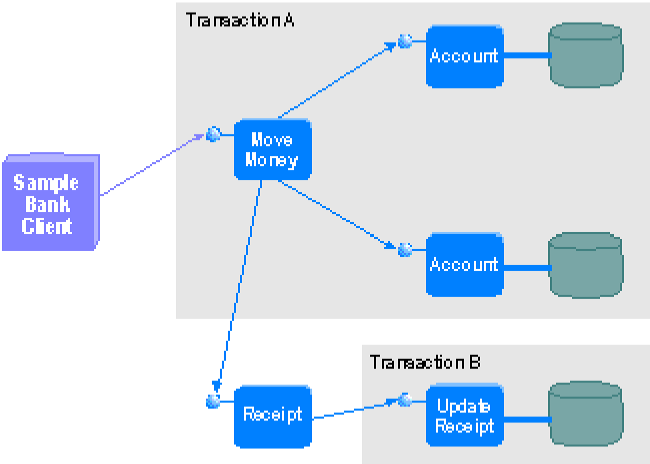
Transactions, Transaction Attributes, Activities

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

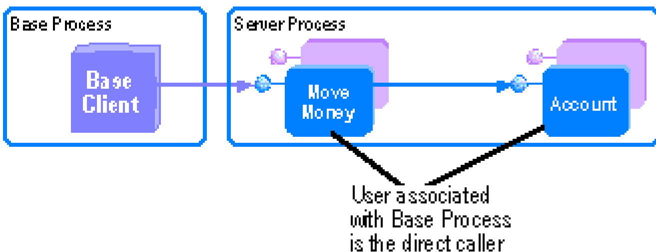
Application Design Notes: Using Roles

This section explains how roles are useful in building secured components. It also discusses how deployment configuration determines security boundaries.

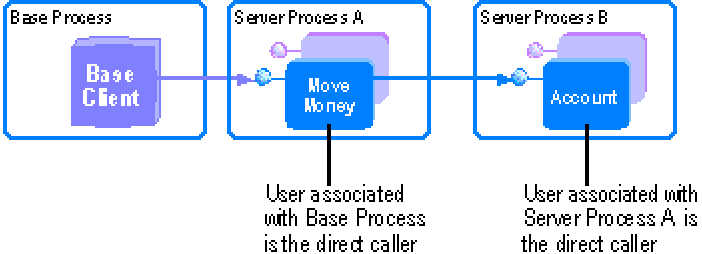
Roles

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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.

Shared Property Group Manager

Shared Property Groups

Shared Property

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

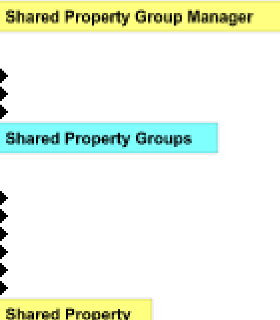
MTS Supported Variant Types, Sharing State

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

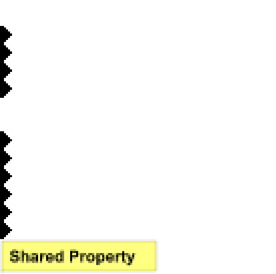
ISharedPropertyGroupManager.getGroup Method, **ISharedPropertyGroup Interface**

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.

Shared Property Group Manager

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

TransactionContextExobject.

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( )**

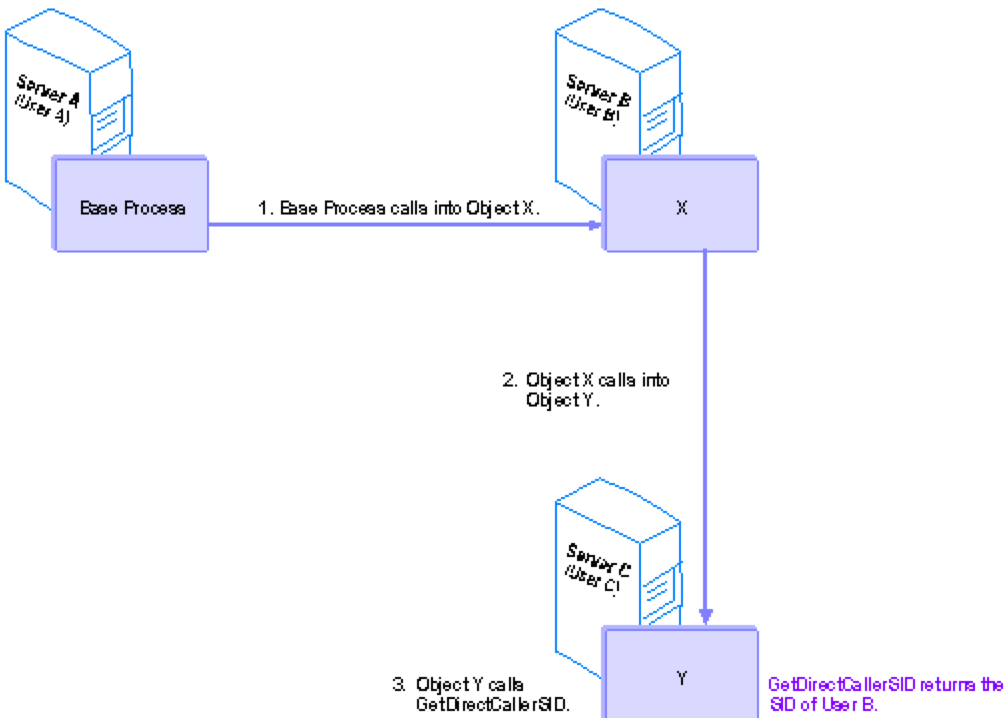
Part

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

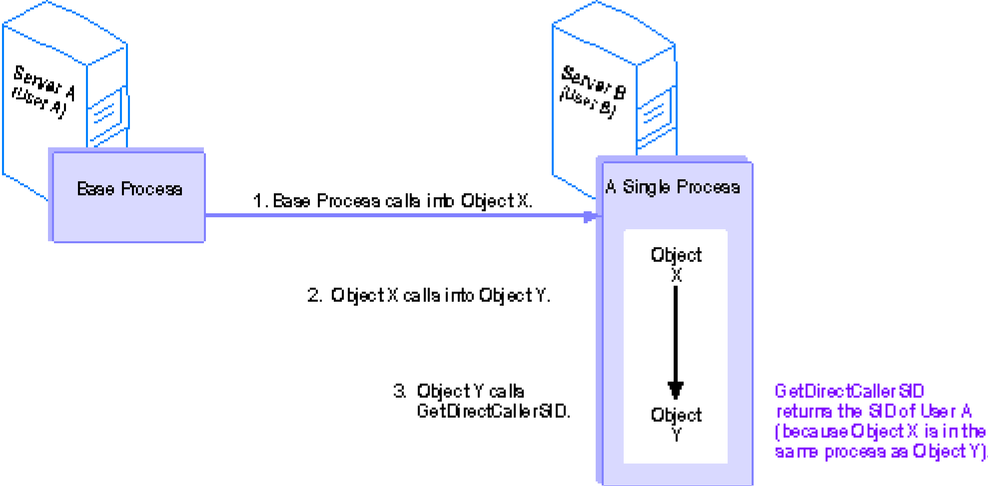
You use the **GetDirectCallerName** method to determine the user name associated with the process that called the object's currently executing method. The following scenarios illustrate the functionality of the **GetDirectCallerName** method.



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( )**

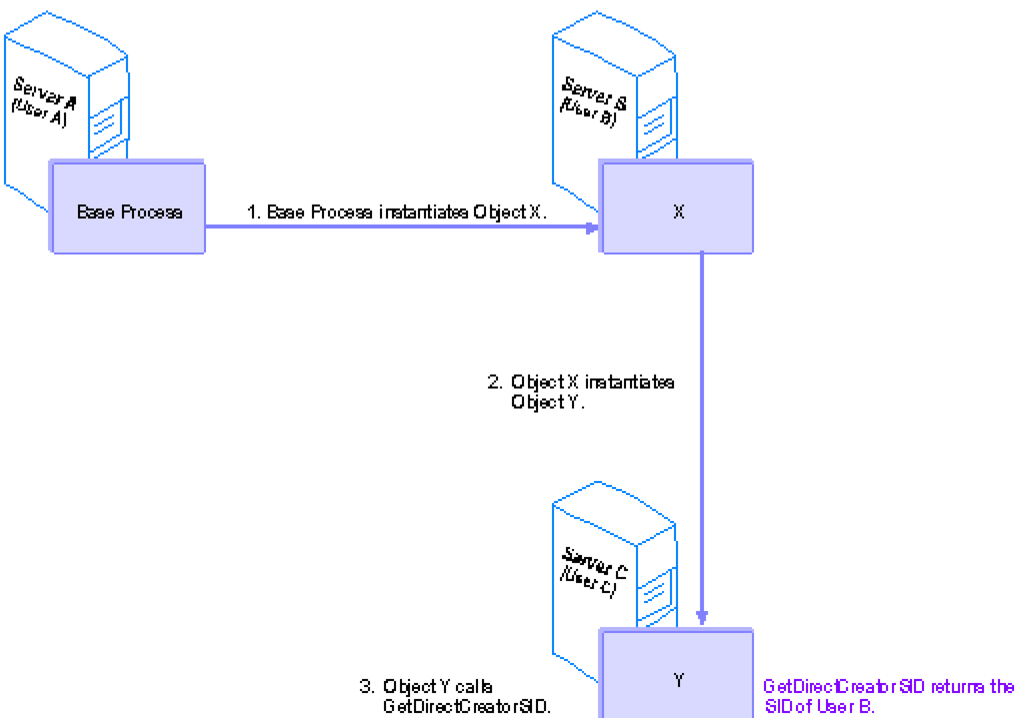
Part

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

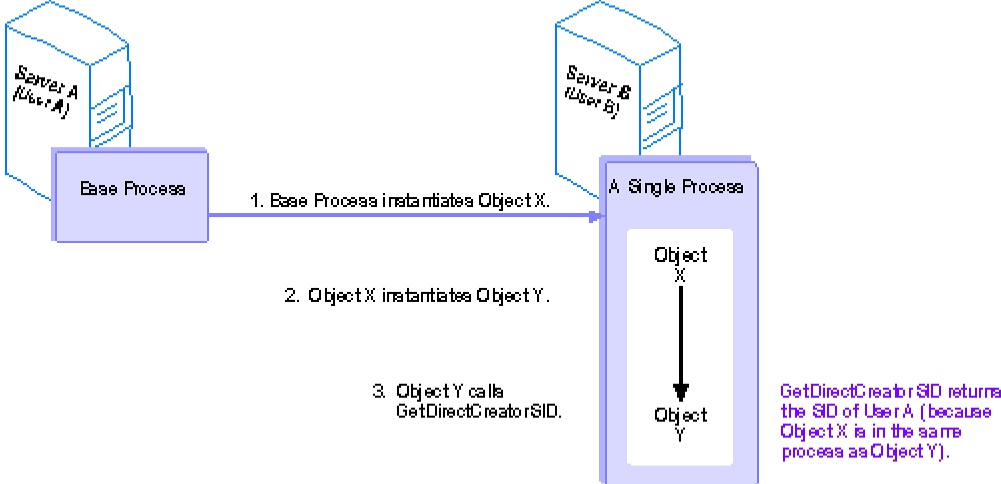
You use the **GetDirectCreatorName** method to determine the user name associated with the process that created the current object. The following scenarios illustrate the functionality of the **GetDirectCreatorName** method.



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( )**

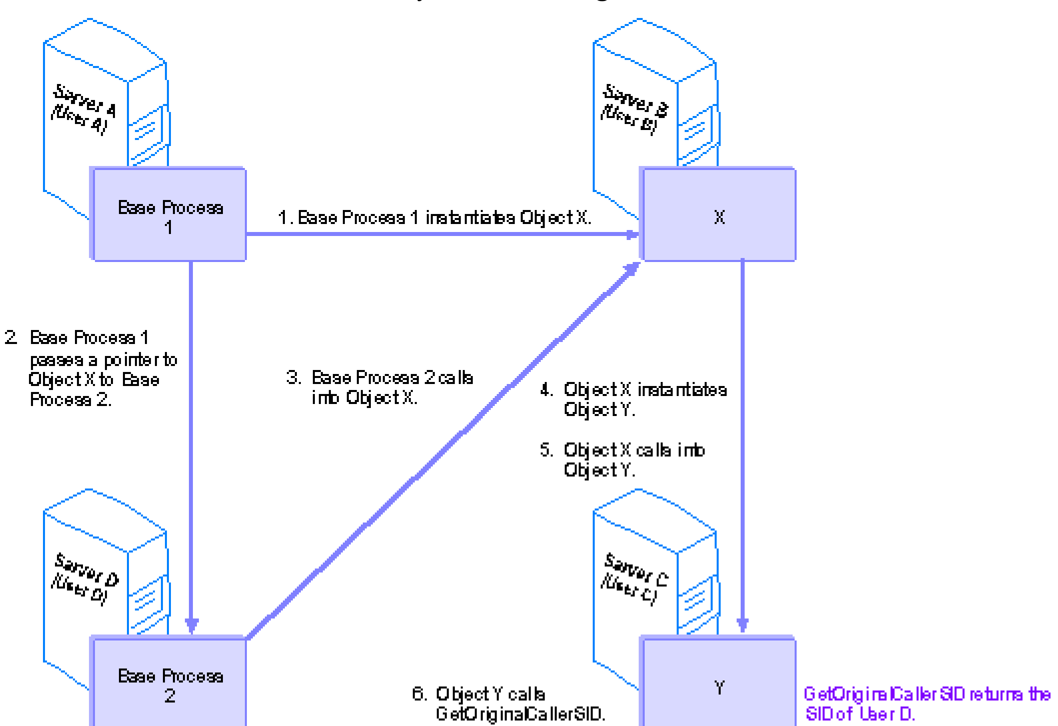
Part

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( )**

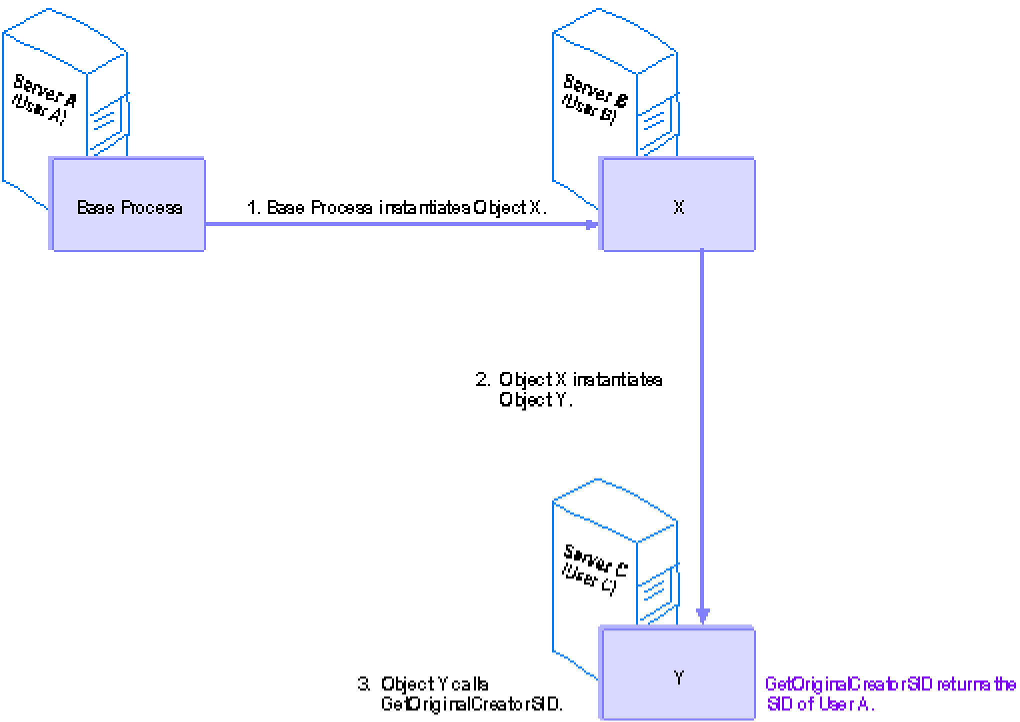
Part

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

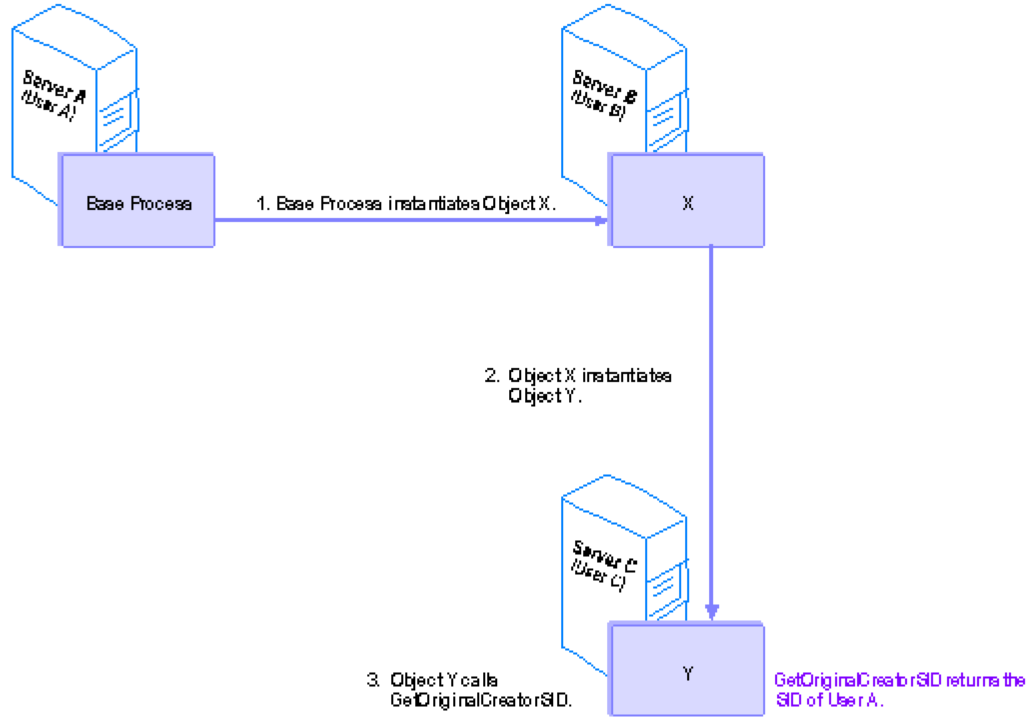
S_OK
  The security ID of the original creator is returned in the parameter *ppSid*.

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
```

Create a New Object

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asNewC;ashowaddcomponents;ashowaddobjects"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asNewS"}

Adds items to the Microsoft Transaction Server Explorer window. The impact of the **New** command is determined by which folder you are currently viewing in the right pane. For example, when the Computers folder is open, clicking **New** adds a new computer.

Toolbar shortcut:

To learn how to add a computer to your Computers folder, see the Configuring Your MTS Deployment Server topic.
To learn how to create a new package, see the Creating an Empty Package topic.
To learn how to create a new role, see the Adding a New Role topic.

Large Icons

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asLargeIconsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asLargeIconsS"}

Displays the items in the right pane of the hierarchy view in their larger format.

Toolbar shortcut:

Small Icons

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asSmallIconsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asSmallIconsS"}

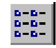
Displays the items in the right pane of the hierarchy view in their smaller format.

Toolbar shortcut:

List View

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asListviewC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asListviewS"}

Displays the items in the right pane of the hierarchy view in their list format.

Toolbar shortcut: 

Property View

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asPropertiesListC;ashowComponentProperties;ashowComputerProperties;ashowInterfaceProperties;ashowPackageProperties;ashowRoleProperties;asProperties"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asPropertiesListS"}

Displays the property settings for the items in the currently selected folder. The properties are displayed in a column format in the Microsoft Transaction Server Explorer's right pane.

Toolbar shortcut:

Which properties are displayed depends on the type of object that is selected. The following table summarizes the properties that are displayed for the different folders.

Folder	Properties
Computers	**Name** The name that has been assigned to the computer and that is recognized within a Windows NT domain. The computer where you are running Transaction Server, however, is referred to as My Computer.
	Timeout
Packages Installed	**Name** The name that you have assigned to a package.
	Security Indicates if security has been enabled for the package.
	Authentication The level of authentication checks.
	Shutdown The period of time before the package shuts down when there is no activity.
	Run Always Indicates the package doesn't shut down when inactive.
	Account The Windows NT account that has been set for the package identity.
	Package ID The Universally Unique Identifier that is assigned to a package.
Components	**Prog ID** The name that has been assigned to a component.
	Transaction Indicates whether the component supports transactions.
	DLL
	CLSID
	Use MTX Indicates whether the component runs in the Transaction Server environment.
	In Process Indicates that the component runs in a different server process.
	Local Indicates that the component runs in a server process on the local computer.
	Remote Indicates that the component runs in a server process on a remote computer.
	Server The name of the remote computer the component runs on.
	Threading The component's threading model.
	Security Indicates whether security has been enabled for the component.

Roles	**Name** The name that has been assigned to the role.
	Role ID The unique identifier that has been assigned to the role.
Interfaces	**Name** The name that has been assigned to the interface.
	Interface ID The unique identifier that has been assigned to the interface.
	Proxy DLL
	TypeLib File The name of the file that contains the type library.
Methods	**Name** The name that has been assigned to the method.
Role Membership	**Name** The name that has been assigned to the role that has been added to a component or interface.
	Role ID The unique identifier that has been assigned to a role.

Status View

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asStatusC;ashowRunandMonitortheAccountComponent"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asStatusS"}

Displays the current state of a component or computer.

Toolbar shortcut:

Computer Status

- **Name** The name that has been assigned to a computer.
- **DTC** Indicates whether MS DTC has been started.

Component Status

- **Prog ID** The name that has been assigned to a component.
- **Objects** The total number of objects that have been allocated within the server process.
- **Activated** The total number of objects that are being used by clients.
- **In Call** The total number of objects that are currently executing a client call.

Refresh Command (View Menu)

{ewc HLP95EN.DLL,DYNALINK,"See Also":"asRefreshC;asRefreshAllComponentsTools"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"asRefreshS"}

Manually updates the information displayed in the Microsoft Transaction Server Explorer's right pane.

Toolbar shortcut:

Glossary

ACID
ActiveX
activity
administrator
aggregation
apartment thread
application executable utility
atomicity
authentication
automatic transaction
base client
base process
Boolean
business rule
caller
catalog
class
class factory
class ID (CLSID)
client
client/server
cluster
COM (Component Object Model)
component
concurrency
consistency
constructor
context
context object properties
creator
data source name (DSN)
deadlock
declarative security
direct caller
direct creator
distributed COM (DCOM)
domain

durability
dynamic-link library (DLL)
exception
failfast
fault isolation
fault tolerance
global account
group
identity
in-doubt transaction
in-process component
instance
interface
isolation
interactive logon user
just-in-time activation
library package
load balancing
local account
main thread
marshaling
method
Microsoft Distributed Transaction Coordinator (MS DTC)
Microsoft Transaction Server component
Microsoft Transaction Server Explorer
Microsoft Transaction Server object
Null
object
object variable
ODBC resource dispenser
OLE Transactions
Open Database Connectivity (ODBC)
original caller
original creator
out-of-process component
package
package file
pooling
pre-built package
process isolation

programmatic identifier (progID)

programmatic security

proxy

remote component

Remote Procedure Call (RPC)

replication

resource dispenser

Resource Dispenser Manager

resource manager

role

safe reference

security ID (SID)

semaphore

server process

server package

shared property

snap-in

stateful object

stateless object

string expression

stub

thread

trace message

transaction

transaction context

transaction manager

transaction timeout

two-phase commit

type library

user name

XA protocol

ACID

The basic transaction properties of atomicity, consistency, isolation, and durability.

ActiveX

A set of technologies that enables software components to interact with one another in a networked environment, regardless of the language in which they were created. ActiveX is built on the Component Object Model (COM).

activity

A collection of Microsoft Transaction Server objects that has a single distributed logical thread of execution. Every Microsoft Transaction Server object belongs to one activity.

administrator

A user that uses the Microsoft Transaction Server Explorer to install, configure, and manage Microsoft Transaction Server components and packages.

aggregation

A composition technique for implementing component objects whereby a new object can be built using one or more existing objects that support some or all of the new object's required interfaces.

apartment thread

A thread used to execute calls to objects of components configured as "apartment threaded." Each object "lives in an apartment" (thread) for the life of the object. All calls to that object execute on the apartment thread. This threading model is used, for example, for component implementations that keep object state in thread local storage (TLS). A component's objects can be distributed over one or more apartments. See also main thread.

application executable utility

A feature in the MTS Explorer that allows you to create an application executable by exporting a package.

atomicity

A feature of a transaction that indicates that either all actions of the transaction happen or none happen.

authentication

The process of determining the identity of a user attempting to access a system. For example, passwords are commonly used to authenticate users.

automatic transaction

A transaction that is created by the Microsoft Transaction Server run-time environment for an object based on the component's transaction attribute.

Boolean

A true/false or yes/no value.

base client

A client that runs outside the Microsoft Transaction Server run-time environment, but that instantiates Microsoft Transaction Server objects.

base process

An application process in which a base client executes. A base client runs outside the Microsoft Transaction Server run-time environment, but instantiates Microsoft Transaction Server objects.

business rule

The combination of validation edits, logon verifications, database lookups, policies, and algorithmic transformations that constitute an enterprise's way of doing business. Also known as *business logic*.

caller

A client that invokes a method of an object. An object's caller isn't necessarily the object's creator. For example, client A could create object X and pass this reference to client B, and then client B could use that reference to call a method of object X. In this case, client A is the creator, and client B is the caller. See also creator.

catalog

The Microsoft Transaction Server data store that maintains configuration information for components, packages, and roles. You can administer the catalog by using the Microsoft Transaction Server Explorer.

class

A type that defines interfaces of a particular type of object. A class defines the properties of the object and the methods used to control the object's behavior.

class factory

An object that implements the **IClassFactory** interface, which allows it to create objects of a specific class.

class ID (CLSID)

A universally unique identifier (UUID) that identifies a COM component. Each COM component has its CLSID in the Windows Registry so that it can be loaded by other applications.

client

An application or process that requests a service from some process or component.

client/server

A distributed application model in which client applications request services from a server application. A server can have many clients at the same time, and a client can request data from multiple servers. An application can be both a client and a server.

cluster

Two or more independent computer systems that are addressed and managed as a single system using Microsoft Cluster Server.

COM (Component Object Model)

An open architecture for cross-platform development of client/server applications based on object-oriented technology. Clients have access to an object through interfaces implemented on the object. COM is language neutral, so any language that produces ActiveX components can also produce COM applications.

component

A discrete unit of code built on ActiveX technologies that delivers a well-specified set of services through well-specified interfaces. Components provide the objects that clients request at run time.

concurrency

The appearance of simultaneous execution of processes or transactions by interleaving the execution of multiple pieces of work.

consistency

A state where durable data matches the state expected by the business rules that modified the data.

constructor

In C++ and Java, a special initialization function that is called automatically whenever an instance of a class is declared. This function prevents errors that result from the use of unitialized objects. The constructor has the same name as the class itself and can't return a value.

context

State that is implicitly associated with a given Microsoft Transaction Server object. Context 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. An object's context is similar in concept to the process context that an operating system maintains for an executing program. The Microsoft Transaction Server run-time environment manages a context for each object.

context object properties

Properties which can be obtained from the context object, such as Internet Information Server intrinsic objects.

creator

A client that creates an object provided by a component (using **CreateObject**, **CoCreateInstance**, or the **CreateInstance** method). When a client creates an object, it is given an object reference that can be used to call the methods of that object. See also caller.

data source name (DSN)

The name that applications use to request a connection to an ODBC data source.

deadlock

A situation in which two or more threads are permanently blocked (waiting), with each thread waiting for a resource exclusively held by one of the other threads that is blocked. For example, if thread A locks record 1 and waits to lock record 2, while thread B has locked record 2 and waits to lock record 1, the two threads are deadlocked.

declarative security

Security that is configured with the Microsoft Transaction Server Explorer. You can control access to packages, components, and interfaces by defining roles. Roles determine which users are allowed to invoke interfaces in a component. See also programmatic security.

direct caller

The identity of the process (base client or server process) calling into the current server process.

direct creator

The identity of the process (base client or server process) that directly created the current object.

distributed COM (DCOM)

DCOM is an object protocol that enables ActiveX components to communicate directly with each other across a network. DCOM is language neutral, so any language that produces ActiveX components can also produce DCOM applications.

domain

In Windows NT, a collection of computers defined by the administrator of a Windows NT server network that share a common directory database. A domain provides access to the centralized user accounts and group accounts maintained by the domain administrator. Each domain has a unique name.

durability

A state that survives failures.

dynamic-link library (DLL)

A file that contains one or more functions that are compiled, linked, and stored separately from the processes that use them. The operating system maps the DLLs into the address space of the calling process when the process is starting or while it's running.

exception

An abnormal condition or error that occurs during the execution of a program and that requires the execution of software outside the normal flow of control.

failfast

A policy of Microsoft Transaction Server that facilitates fault containment. When the Transaction Server encounters an unexpected internal error condition, it immediately terminates the process and logs messages to the Windows NT event log for details about the failure.

fault isolation

Containing the effects of a fault within a component, rather than propagating the fault to other components in the system.

fault tolerance

The ability of a system to recover from an error, a failure, or a change in environmental conditions (such as loss of power). True fault tolerance provides for fully automatic recovery without disruption of user tasks or files, in contrast to manual means of recovery such as restoring data loss with backup files.

global account

A normal user account in the user's home domain. Most accounts are global accounts, which is the default setting. If multiple domains are available, it's best if each user in the network has only one global account in only one domain.

group

A name that identifies a set of one or more Windows NT users accounts.

identity

A package property that specifies the user accounts that are allowed to access the package. It can be a specific user account or a group of users within a Windows NT domain.

in-doubt transaction

A transaction that has been prepared but hasn't received a decision to commit or abort because the server coordinating the transaction is unavailable.

in-process component

A component that runs in a client's process space. This is typically a dynamic-link library (DLL).

instance

An object of a particular component class. Each instance has its own private data elements or member variables. A component instance is synonymous with object.

interactive logon user

The user that is currently logged on a Windows Transaction Server computer.

interface

A group of logically related operations or methods that provides access to a component object.

isolation

A characteristic whereby two transactions running in parallel produce the illusion that there is no concurrency. It appears that the system runs one transaction at a time.

just-in-time activation

The ability for a Microsoft Transaction Server object to be activated only as needed for executing requests from its client. Objects can be deactivated even while clients hold references to them, allowing otherwise idle server resources to be used more productively.

library package

A package that runs in the process of the client that creates it. Library packages do not support component tracking, role checking, or process isolation. MTS supports two types of packages: Library package and server package.

load balancing

Distribution of the processing load among several servers carrying out network tasks to increase overall network performance.

local account

An account provided in a local domain for a user whose regular account isn't in a trusted domain. Local accounts cannot be used to log on interactively. Local accounts created in one domain cannot be used in trusted domains.

main thread

A single thread used to run all objects of components marked as "single threaded." See also apartment thread.

marshaling

The process of packaging and sending interface method parameters across thread or process boundaries.

method

A procedure (function) that acts on an object.

Microsoft Distributed Transaction Coordinator (MS DTC)

A transaction manager that coordinates transactions that span multiple resource managers. Work can be committed as an atomic transaction even if it spans multiple resource managers, potentially on separate computers.

Microsoft Transaction Server component

A COM component that executes in the Microsoft Transaction Server run-time environment. A Transaction Server component must be a dynamic-link library (DLL), implement a class factory to create objects, and describe all of the component's interfaces in a type library for standard marshaling.

Microsoft Transaction Server Explorer

An application to configure and manage Microsoft Transaction Server components within a distributed computer network.

Microsoft Transaction Server object

A COM object that executes in the Microsoft Transaction Server run-time environment and follows the Transaction Server programming and deployment model.

Null

A value that indicates missing or unknown data.

object

A run-time instance of a COM component. An object is created by a component's class factory. Object is synonymous with instance.

object variable

A variable that contains a reference to an object.

ODBC resource dispenser

A resource dispenser that manages pools of database connections for Microsoft Transaction Server components that use the standard ODBC programming interfaces.

OLE Transactions

OLE Transactions is an object-oriented, two-phase commit protocol based on the Component Object Model (COM). It is used by resource managers in order to participate in distributed transactions coordinated by Microsoft Distributed Transaction Coordinator (DTC).

Open Database Connectivity (ODBC)

A standard programming language interface used to connect to a variety of data sources.

original caller

The identity of the base client that initiated the activity.

original creator

The identity of the base client that created the current object. The original caller and original creator are different only if the original creator passed the object to another base client. See also original caller.

out-of-process component

A component that runs in a separate process space from its client. The Microsoft Transaction Server enables components implemented as DLLs to be used out-of-process from the client, by loading the components into surrogate server processes.

package

A set of components that perform related application functions. All components in a package run together in the same Microsoft Transaction Server server process. A package is a trust boundary that defines when security credentials are verified, and a deployment unit for a set of components. You can create packages with the Transaction Server Explorer. Packages can be either a library package or server package.

package file

A file that contains information about the components and roles of a package. A package file is created using the package export function of the Transaction Server Explorer. When you create a pre-built package, the associated component files (DLLs, type libraries, and proxy-stub DLLs, if implemented) are copied to the same directory where the package file was created.

pooling

A performance optimization based on using collections of pre-allocated resources, such as objects or database connections. Pooling results in more efficient resource allocation.

pre-built package

A package file that contains information about the components and roles of a package. A package file is created using the package export function of the Transaction Server Explorer. When you create a pre-built package, the associated component files (DLLs, type libraries, and proxy-stub DLLs, if implemented) are copied to the same directory where the package file was created.

process isolation

The technique of running a server process in a separate memory space in order to isolate that process from other server processes. Process isolation protects a server process from other fatal application errors. Isolating a server process also prevents the isolated process from terminating another server process with an application fatal error. An MTS package that supports process isolation is called a Server package.

programmatic identifier (progID)

A name that identifies a COM component. For example, a programmatic ID could be Bank.MoveMoney.

programmatic security

Procedural logic provided by a component to determine if a client is authorized to perform the requested operation. See also declarative security.

proxy

An interface-specific object that provides the parameter marshaling and communication required for a client to call an application object that is running in a different execution environment, such as on a different thread or in another process. The proxy is located with the client and communicates with a corresponding stub that is located with the application object that is being called.

remote component

A component used by a client on a different computer.

Remote Procedure Call (RPC)

A standard that allows one process to make calls to functions that are executed in another process. The process can be on the same computer or on a different computer in the network.

replication

An operation which copies the catalog from one computer to another. Replication is used to synchronize clustered MTS servers.

resource dispenser

A service that provides the synchronization and management of nondurable resources within a process, providing for simple and efficient sharing by Microsoft Transaction Server objects. For example, the ODBC resource dispenser manages pools of database connections.

Resource Dispenser Manager

A dynamic-link library (DLL) that coordinates work among a collection of resource dispensers.

resource manager

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. The resource managers work in cooperation with the transaction manager to provide the application with a guarantee of atomicity and isolation (using the two-phase commit protocol). Microsoft SQL Server is an example of a resource manager.

role

A symbolic name that defines a class of users for a set of components. Each role defines which users are allowed to invoke interfaces on a component.

safe reference

A reference to the current object that is safe to pass outside the current object's context.

security ID (SID)

A unique name that identifies a logged-on user to the security system. SIDs can identify one user or a group of users.

semaphore

A locking mechanism used inside resource managers or resource dispensers. Semaphores have no symbolic names, only shared and exclusive mode access, no deadlock detection, and no automatic release or commit.

server package

A package that runs isolated in its own process on the local computer. Server packages support role-based security, resource sharing, process isolation, and process management (such as package tracking). MTS supports two types of packages: library and server package.

server process

A process that hosts Microsoft Transaction Server components.

A Microsoft Transaction Server component can be loaded into a surrogate server process, either on the client's computer or into a client application process.

shared property

A variable that is available to all objects in the same server process via the Shared Property Manager. The value of the property can be any type that can be represented by a variant.

snap-in

An administrative program hosted by the Microsoft Management Console (MMC). The MTS Explorer on Windows NT is a snap-in.

stateful object

An object that holds private state accumulated from the execution of one or more client calls.

stateless object

An object that doesn't hold private state accumulated from the execution of one or more client calls.

string expression

Any expression that evaluates to a sequence of contiguous characters.

stub

An interface-specific object that provides the parameter marshaling and communication required for an application object to receive calls from a client that is running in a different execution environment, such as on a different thread or in another process. The stub is located with the application object and communicates with a corresponding proxy that is located with the client that calls it.

thread

The basic entity to which the operating system allocates CPU time. A thread can execute any part of the application's code, including a part currently being executed by another thread. All threads of a process share the virtual address space, global variables, and operating-system resources of the process.

transaction

A unit of work that is done as an atomic operation—that is, the operation succeeds or fails as a whole.

transaction context

An object used to allow a client to dynamically include one or more objects in one transaction.

transaction manager

A system service responsible for coordinating the outcome of transactions in order to achieve atomicity. The transaction manager ensures that the resource managers reach a consistent decision on whether the transaction should commit or abort.

trace message

A message that includes the current status of various Microsoft Transaction Server activities, such as startup and shutdown.

transaction timeout

The maximum period of time that a transaction can remain active before it's automatically aborted by the transaction manager.

type library

A file containing standard descriptions of data types, modules, and interfaces that can be used to fully expose objects with ActiveX technology.

two-phase commit

A protocol that ensures that transactions that apply to more than one server are completed on all servers or none at all. Two-phase commit is coordinated by the transaction manager and supported by resource managers.

user name

The name that identifies a Windows NT user account.

XA protocol

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.