TIBCO® Adapter SDK

Programmer's Guide

Software Release 5.8
November 2011

TIBCO provides the two-second advantage™

TIBCO®
The Power of Now®

Important Information

Contents

Figures

Tables

Preface

TIBCO Adapter SDK (Software Development Kit) is a class library that facilitates adapter development. All adapters implemented using TIBCO Adapter SDK have the same external interface and consistently plug in to the overall TIBCO ActiveEnterprise product suite.

Topics

Changes from the Previous Release of this Guide

This section itemizes the major changes from the previous release of this guide.

MSVC++ 8 SP1

The MSVC++ 8 SP1 compiler is supported on Windows platforms in this release. Refer to Requirements on Microsoft Windows on page 2 for more information.

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Adapter SDK Documentation

The following documents form the TIBCO Adapter SDK documentation set:

- *TIBCO Adapter SDK Concepts* Read this manual before reading any other book in the documentation set to familiarize yourself with the product and its use.
- *TIBCO Adapter SDK Installation* Read this manual for instructions on site preparation and installation.
- *TIBCO Adapter SDK Programmer's Guide* Read this manual for details on implementing a custom adapter. This manual also discusses configuration and programming, and provides example code fragments.
- *TIBCO Adapter SDK Status Codes* A reference for the message codes used by TIBCO Adapter SDK.
- *TIBCO API Reference* Provides online documentation for the exposed interfaces, classes, and methods of the TIBCO Adapter C++ and Java APIs.
- *TIBCO Adapter SDK Release Notes* Read the release notes for a list of new and changed features. This document also contains lists of closed and known issues for this release.

Other TIBCO Product Documentation

You may find it useful to read the documentation for the following TIBCO products:

- TIBCO ActiveMatrix BusinessWorks™
- TIBCO ActiveEnterprise™
- TIBCO Designer™
- TIBCO Administrator™
- TIBCO Rendezvous®
- TIBCO Enterprise Message Service™
- TIBCO Hawk®
- TIBCO Runtime Agent™

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
ENV_NAME *TIBCO_HOME* *SDK_HOME*	TIBCO products are installed into an installation environment. A product installed into an installation environment does not access components in other installation environments. Incompatible products and multiple instances of the same product must be installed into different installation environments. An installation environment consists of the following properties: • **Name** Identifies the installation environment. This name is referenced in documentation as *ENV_NAME*. On Microsoft Windows, the name is appended to the name of Windows services created by the installer and is a component of the path to the product shortcut in the Windows Start > All Programs menu. • **Path** The folder into which the product is installed. This folder is referenced in documentation as *TIBCO_HOME*. TIBCO Adapter SDK is installed into a directory within a *TIBCO_HOME*. This directory is referenced in documentation as *SDK_HOME*. The default value of *SDK_HOME* depends on the operating system. For example, on Windows systems, the default value is `C:\tibco\adapter\SDK\`*version_number*.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: `MyCommand [`**`enable`**` | disable]`

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO ActiveMatrix BusinessWorks Concepts*. • To introduce new terms. For example: A portal page may contain several portlets. *Portlets* are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *PathName*
Key combinations	Key names separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
📝	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
💡	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
⚠	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2 Syntax Typographical Conventions

Convention	Use
`[ ]`	An optional item in a command or code syntax. For example: `MyCommand [optional_parameter] required_parameter`
`|`	A logical OR that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: `MyCommand para1 | param2 | param3`

Table 2 Syntax Typographical Conventions

Convention	Use

`{ }`	A logical group of items in a command. Other syntax notations may appear within each logical group.
	For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`.
	`MyCommand {param1 param2} | {param3 param4}`
	In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:
	`MyCommand {param1 | param2} {param3 | param4}`
	In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`.
	`MyCommand param1 [param2] {param3 | param4}`

Connecting with TIBCO Resources

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts, a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to http://www.tibcommunity.com.

How to Access All TIBCO Documentation

After you join TIBCOmmunity, you can access the documentation for all supported product versions here:

http://docs.tibco.com/TibcoDoc

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows:

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a username and password. If you do not have a username, you can request one.

Chapter 1 Programming Requirements and TIBCO Adapter SDK Classes

This chapter lists the TIBCO Adapter SDK programming requirements on Microsoft Windows and UNIX systems. This chapter also provides an overview of the major classes in TIBCO Adapter SDK.

Topics

Requirements on Microsoft Windows

Table 3 and Table 4 list the environment variables and shared libraries required for Microsoft Windows.

Table 3 Environment Variables for Microsoft Windows

Variable	Must Include	Example (Command Line)

`PATH`	TIBCO Adapter SDK installation folder's `bin` folders.
	If using TIBCO Enterprise Message Service, include the `ems` entry. If using TIBCO Enterprise for JMS, use `jms` in the path instead of `ems`.
		`SET`
		`PATH=C:\tibco\adapter\sdk\`*version_number*`\bin;C:\tibco\tibrv\`*version_number*`\bin;C:\tibco\tpcl\`*version_number*`\bin;C:\tibco\ems\`*version_number*`\bin;%PATH%`

`INCLUDE`	TIBCO Adapter SDK installation folder's `include` folder.	`SET`
		`INCLUDE=C:\tibco\adapter\sdk\`*version_number*`\include;C:\tibco\tibrv\`*version_number*`\include;C:\tibco\tpcl\`*version_number*`\include;C:\tibco\tpcl\`*version_number*`\include\xercesc;%INCLUDE%`

`LIB`	TIBCO Adapter SDK installation folder's `lib` folder.	`SET`
		`LIB=C:\tibco\adapter\sdk\`*version_number*`\lib;C:\tibco\tibrv\`*version_number*`\lib;C:\tibco\tpcl\`*version_number*`\lib;%LIB%`

The environment variables must also be set appropriately for TIBCO Rendezvous and TIBCO Designer. See the relevant product documentation for more information.

Table 4 Shared Libraries for Microsoft Windows

Item	Description
Supported Platform	Refer to the readme file for the supported platform.
Compiler	MSVC++ 8 SP1
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` files. TIBCO Enterprise Message Service `include` files.

Table 4 Shared Libraries for Microsoft Windows (Cont'd)

Item	Description
Preprocessor symbols	none
SDK library	(32-bit platforms) *SDK_HOME*`/lib/maverick58d.lib` (debug) or *SDK_HOME*`/lib/maverick58.lib` (release) (64-bit platforms) *SDK_HOME*`/lib/64/maverick58d.lib` (debug) or *SDK_HOME*`/lib/64/maverick58.lib` (release)

Requirements on UNIX

For all UNIX systems:

1. Include the TIBCO Adapter SDK header file, `Maverick.h`, in the program.
2. Add the `include` directories of the following products to the `makefile`: TIBCO Adapter SDK, TIBCO Rendezvous, and TIBCO Enterprise Message Service.
3. Set the environment variables as listed in Environment Variables for UNIX Systems on page 4.
4. Compile applications with an ANSI-compliant C++ compiler. See Compiling Requirements for UNIX Systems on page 5.

Environment Variables for UNIX Systems

To compile the example programs, the C++ examples included in the package, you can use the `configure` tool, which is also included, to generate a makefile. Makefiles generated by the `configure` tool set the following environment variables.

Table 5 Environment Variables for UNIX Systems

Variable	Set to...
RV_ROOT	Directory where TIBCO Rendezvous is installed.
TRA_ROOT	Directory where TIBCO Runtime Agent is installed.
SDK_ROOT	Directory where TIBCO Adapter SDK is installed.
JMS_ROOT	Directory where TIBCO Enterprise Message Service is installed.
TPCL_ROOT	Directory where third-party client library is installed.
CCC	Compiler location if CC is not in `$PATH`.
CFLAGS	Compiler flags, if defined, override configure-generated flag.
USER_DEF_PATH	If special `include` and `lib` paths are needed.

Compiling Requirements for UNIX Systems

This section introduces the compiler, `includes`, preprocessor symbols for the individual platforms.

- Solaris Development on SPARC, page 5
- Solaris Development on X86, page 6
- HP-UX Development on PA-RISC, page 8
- HP-UX 11iv2, 11iv3 Development on Itanium 2, page 9
- AIX Development, page 10
- Linux Development, page 11

The sequence in which the link libraries are specified is important for some platforms.

Solaris Development on SPARC

Table 6 Solaris Development on SPARC

Item	Description	Compiler flags
Solaris (SPARC 32-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	CC 5.9	`-mt`
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick58`
	`-ltibrvft`
	`-ltibrvcmq`
	`-ltibrvcm`
	`-ltibrv`
	`-lsocket`
	`-lgen`
	`-lnsl`
	`-ldl`
	`-lrt`
	`-lrepowww580`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Table 6 Solaris Development on SPARC (Cont'd)

Item	Description	Compiler flags
Solaris (SPARC 64-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	CC 5.9, CC 5.10	`-mt` `-m64`
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick5864`
	`-ltibrvft64`
	`-ltibrvcmq64`
	`-ltibrvcm64`
	`-ltibrv64`
	`-lsocket`
	`-lgen`
	`-lnsl`
	`-ldl`
	`-lrt`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Solaris Development on X86

Table 7 Solaris Development on X86

Item	Description	Compiler flags
Solaris (x86)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	CC 5.9 or Sun Studio 12	`-mt`
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Table 7 Solaris Development on X86 (Cont'd)

Item	Description	Compiler flags

Libraries	`-lmaverick58`
	`-ltibrvft`
	`-ltibrvcmq`
	`-ltibrvcm`
	`-ltibrv`
	`-lsocket`
	`-lgen`
	`-lnsl`
	`-ldl`
	`-lrt`
	`-lrepowww580`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Solaris (x86-64)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	CC 5.9 or Sun Studio 12	`-mt` `-m64`
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick5864`
	`-ltibrvft64`
	`-ltibrvcmq64`
	`-ltibrvcm64`
	`-ltibrv64`
	`-lsocket`
	`-lgen`
	`-lnsl`
	`-ldl`
	`-lrt`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

HP-UX Development on PA-RISC

Table 8 HP-UX Development on PA-RISC

Item	Description	Preprocessor symbols
HP (PA-RISC 32-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	

Compiler	aCC 3.13	`-DHP_UX`
		`-D_REENTRANT`
		`+DAportable`

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick58`
	`-ltibrv`
	`-ltibrvcm`
	`-ltibrvcmq`
	`-ltibrvft`
	`-lrt`
	`-lrepowww580`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`
	`-lpthread`	

HP (PA-RISC 64-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	

Compiler	aCC 3.13	`-DHP_UX`
		`-D_REENTRANT`
		`+DA2.0W`

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Table 8 HP-UX Development on PA-RISC (Cont'd)

Item	Description	Preprocessor symbols

Libraries	`-lmaverick5864`
	`-ltibrv64`
	`-ltibrvcm64`
	`-ltibrvcmq64`
	`-ltibrvft64`
	`-lrt`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`
	`-lpthread`	

HP-UX 11iv2, 11iv3 Development on Itanium 2

With HP-UX 11iv2 and 11iv3 on Itanium, you need the following environmment.

Table 9 HP-UX 11iv2, 11iv3 Development on Itanium 2

Item	Description	Complier flags
HP-UX on Itanium 2 (32-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	

Compiler	`aCC A.05.50`	`-mt`
		`+DAportable -DHP_UX`
		`-D_REENTRANT`

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick58`
	`-lxerces-c2_8`
	`-lrepowww580`
	`-ltibrv`
	`-ltibrvcm`
	`-ltibrvcmq`
	`-ltibrvft`
	`-lrt`
	`-lpthread`
	`-lssl`
	`-lcrypto`	

HP-UX on Itanium 2 (64-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	

Table 9 HP-UX 11iv2, 11iv3 Development on Itanium 2 (Cont'd)

Item	Description	Complier flags

Compiler	`aCC A.05.50`	`+DD64`
		`-AA`
		`-DHP_UX -D_REENTRANT`
		`-DHPUX_IA64`
		`-D_THREAD_SAFE -mt`

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick5864`
	`-ltibrv64`
	`-ltibrvcm64`
	`-ltibrvcmq64`
	`-ltibrvft64`
	`-lrt`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`
	`-lpthread`	

AIX Development

With AIX you need the following environment.

Table 10 AIX Development

Item	Description	Compiler flags
AIX (32-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	xlC_r v5.1.0 xlC_r v8.0 with `-qnamemangling=v5` xlC_r v9.0 with `-qnamemangling=v5` xlC_r v11.0 with -U__STR__ and `-qnamemangling=v5`	-+ -qcpluscmt -qroconst -qproto -qchars=signed-qlongdouble -qstaticinline -DU_SIZEOF_WCHAR_T=2
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Table 10 AIX Development (Cont'd)

Item	Description	Compiler flags

Libraries	`-lmaverick58`
	`-ltibrv`
	`-ltibrvcm`
	`-ltibrvcmq`
	`-ltibrvft`
	`-lrepowww580`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

AIX (64-bit)		
Supported Platforms	Refer to the readme file for the supported versions.	
Compiler	xlC_r v8.0 with `-qnamemangling=v5` xlC_r v9.0 with `-qnamemangling=v5` xlC_r v11.0 with -U__STR__ and `-qnamemangling=v5`	-+ -qcpluscmt -qroconst -qproto -qchars=signed-qlongdouble -qstaticinline -DU_SIZEOF_WCHAR_T=4
Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick5864`
	`-ltibrv64`
	`-ltibrvcm64`
	`-ltibrvcmq64`
	`-ltibrvft64`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Linux Development

With Linux, you need the following environment.

Table 11 Linux Development

Item	Description	Compiler flags
Linux (x86)		
Supported Platforms	Refer to the readme file for the supported versions.	

Table 11 Linux Development (Cont'd)

Item	Description	Compiler flags

Compilers	GNU gcc 3.2.3, gcc 4.4.1	`-pthread`
		`-fPIC`
		Note: Add "`-DLINUX24`" to `CCFLAGS` option in the makefile.

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick58`
	`-ltibrv`
	`-ltibrvcm`
	`-ltibrvcmq`
	`-ltibrvft`
	`-ldl`
	`-lrepowww580`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Linux (x86-64)		
Supported Platforms	Refer to the readme file for the supported versions.	

Compilers	GNU gcc 3.4.4, gcc 4.4.1	`-pthread`
		`-fPIC`
		`-m64`
		Note: Add "`-DLINUX24`" to `CCFLAGS` option in the makefile.

Includes	TIBCO Adapter SDK directory. TIBCO Rendezvous `include` directory.	

Libraries	`-lmaverick5864`
	`-ltibrv64`
	`-ltibrvcm64`
	`-ltibrvcmq64`
	`-ltibrvft64`
	`-ldl`
	`-lrepowww58064`
	`-lxerces-c2_8`
	`-lssl`
	`-lcrypto`	

Java SDK Requirements

To compile and run a Java SDK application, you need to set up the `CLASSPATH` and `PATH` (or `LD_LIBRARY_PATH` on UNIX). The shared library path must be set because the TIBCO Rendezvous Java implementation uses JNI to access its shared libraries at runtime. If the TIBCO Rendezvous shared library path is not specified, a runtime exception of `Native Implementation required` is thrown.

Assuming that the TIBCO Runtime Agent (with SDK suite) has been installed in `C:\tibco\tra`, TPCL in `C:\tibco\tpcl`, Rendezvous in `C:\tibco\tibrv\8.1`, and JDK in `C:\jdk`, the `CLASSPATH` and `PATH` should be set as follows:

```
set CLASSPATH=\
   .;\
   C:\tibco\tra\version_number\hotfix\lib\TIBCOrt.jar;\
   C:\tibco\tra\version_number\lib\TIBCOrt.jar;\
   C:\tibco\tra\version_number\lib\TIBCOxml.jar;\
   C:\tibco\tra\version_number\lib\TIBCrypt.jar;\
   C:\tibco\tpcl\version_number\lib\xmlParserAPIs.jar;\
   C:\tibco\tpcl\version_number\lib\xercesImpl.jar;\
   C:\tibco\tibrv\version_number\lib\tibrvjsd.jar
```

```
set PATH=
   C:\jdk\bin;
   C:\tibco\tibrv\version_number\bin;
   C:\tibco\tra\version_number\hotfix\bin;
   C:\tibco\tra\version_number\bin;
   %PATH%
```

For TIBCO Enterprise Message Service, include the following JAR files in the classpath. For TIBCO Enterprise for JMS, use `jms` in the path, not `ems`.

```
C:\tibco\ems\version_number\lib\tibjms.jar
C:\tibco\ems\version_number\lib\jms.jar
```

Remember to add all hotfix JAR files (if there are any) under the *SDK_HOME*`\hotfix\lib` directory ahead of the `CLASSPATH`.

Note on SSL

Generally speaking, a security provider JAR must be part of `CLASSPATH` whenever encryption or secure connections are used. Specific scenarios are:

- If SSL is configured for adapter message transport (JMS or Rendezvous).
- If an adapter connects to *security enabled* TIBCO Administrator Server (password check or full `https://` connection).
- If the local file repository contains an encrypted global variable that requires a security provider to decrypt.

For example, if your applications require SSL and you are using Entrust, ensure that the `CLASSPATH` includes the Entrust JAR file. See *install-path*`\tibco\tpcl\`*version_number*`\lib\entrust`.

```
set TPCL_ROOT=C:\tibco\tpcl\version_number
set
CLASSPATH=%CLASSPATH%;%TPCL_ROOT%\lib\entrust;%TPCL_ROOT%\lib\entr
ust\enttoolkit.jar;
```

To use a different security vendor, change the `TIBCO_SECURITY_VENDOR` java property. For example, if you use J2SE as the vendor then you must either:

- specify `java.property.TIBCO_SECURITY_VENDOR=j2se` in the tra property file or
- add `-DTIBCO_SECURITY_VENDOR=j2se` in the command line

In addition, vendor libraries must be included in the `CLASSPATH` (this is automatically so for `j2se`).

TIBCO Adapter SDK Classes

Table 12 lists the major classes in the TIBCO Adapter SDK and the location in this document where they are discussed. For additional information, see the API Reference available online.

Table 12 TIBCO Adapter SDK Classes

Class	Description	See
`MApp`	Handles initialization and shutdown. Manages other elements and dispatches messages to other elements' methods.	MApp Application Manager, page 45
`MRvSession` `MJmsSession`	Support instantiation of sessions.	MApp Application Manager, page 45
`MDispatcher`	Supports multi threaded SDK adapters.	Multithreaded Adapters, page 124
`MTree`	Represents data in a heterogeneous format.	Adapter Application Data, page 66
`MData` and subclasses (`MInstance`, `MSequence`, `MUnion`, and data encapsulation classes)	Represent data constrained by external metadata descriptions. Different classes exist for different types.	Metadata, page 89
`MClassRegistry` and related classes	Provide access to metadata information.	Metadata, page 89
`MMetaDescription` and subclasses	Metadata class descriptions including SDK-based and custom types.	Metadata, page 89

`MEventSource`
`MEvent`
`MEventListener`
	Define event model.	Event Model, page 60

`MPublisher` `MSubscriber`	Send or receive data using the chosen message format.	Transport Protocol, page 51

Table 12 TIBCO Adapter SDK Classes (Cont'd)

Class	Description	See

`MProperties`
`MPropertiesRegistry`
`MConfigurationUtilities`
	Define processing and retrieval of configuration information and other structured information that is not application data.	Adapter Configuration, page 17

`MTrace` `MSink` (and related classes)	Define tracing (logging) behavior.	Tracing, page 72
`MException`	Defines exceptions.	Exception Handling, page 83
`MAdvisory`	Defines advisories (usually TIBCO Rendezvous advisories).	Advisory Handling, page 138
`MMessageBundle`	Encapsulates trace message information.	Using Tracing Facilities, page 72
`MTrackingInfo`	Allows tracking of messages across the ActiveEnterprise.	Tracking, page 81
`MHawkMicroagent`	Allows easy integration with TIBCO Hawk AMI (Application Management Interface).	TIBCO Adapters and TIBCO Hawk, page 153
`MOperation` and related classes	Implement AEOperation invocation.	TIBCO ActiveEnterprise Operation Model, page 113

Chapter 2 Adapter Configuration

This chapter gives an introduction to TIBCO Designer and explains how to create, modify, and save project repositories. It also includes a list of the standard command-line arguments for C++ and Java TIBCO Adapter SDK adapters, as well as instructions on how to supply certain configuration information from the command line or in a properties file.

Topics

Overview

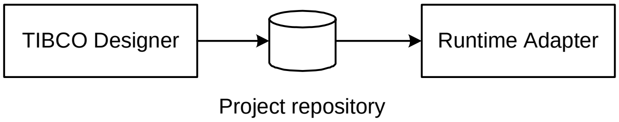
TIBCO Designer is available for configuring SDK-based custom adapters. Adapter configurations are saved to the project repository, where the code can access them.

In general, an adapter developer needs to go through the following steps:

1. Prepare the adapter configuration using TIBCO Designer and export it to the project repository. See Configuring and Exporting the Project Repository on page 20.
2. Write the custom adapter code. Ensure that the project repository is known to the adapter. The runtime adapter program accesses the information in the project repository.

   The repository location (as well as some startup information) can be specified on the command line, in a properties file, or in the adapter program itself. See Specifying Configuration Information on page 22.
3. Upon startup, the SDK encapsulates the configuration information in instances of classes. The custom adapter code can then access that information. See Accessing Configuration Information on page 27.

Figure 1 Adapter Configuration Overview



Types of Configuration Information

The first step in developing a custom adapter is to specify the configuration information. TIBCO Adapter SDK allows you to specify various types of configuration information:

Application Objects (publisher, subscriber, session, and so on)

Application objects are specified in TIBCO Designer using the `Generic Adapter Configuration` resource and saved in the project repository.

At runtime, the SDK accesses the descriptions based on the `configUrl` or `repoUrl` and stores them in the `MProperties` class. The custom adapter can then access the data through `MProperties`. See Sending and Receiving Data on page 65.

Metadata Information (class description and operation description)

Metadata information is specified using TIBCO Designer or by editing the AEXML repository file.

At runtime, the SDK accesses the descriptions based on the `configUrl` or `repoUrl` and creates class description classes (for example, `MClassDescription`). Custom adapters access the class descriptions through the `MClassRegistry` object and create instances based on the description. See Metadata on page 89.

Global Variables

Global variables are specified in TIBCO Designer, in a properties file, or on the command-line. See Variable Substitution on page 34.

Startup Information

Startup information is specified using the `Generic Adapter Configuration` resource inside the application code, in a properties file, or on the command-line.

Configuring and Exporting the Project Repository

Configuration and metadata information for an adapter instance are specified in TIBCO Designer.

The *TIBCO Designer Palette Reference* explains how to use TIBCO Designer for adapter configuration and gives a reference to the configuration objects and attributes. It also explains how to use TIBCO Designer for schema configuration and gives a reference to the schema objects and attributes. In TIBCO Designer, click **What is This** on any resource for help information.

Configuring an Adapter

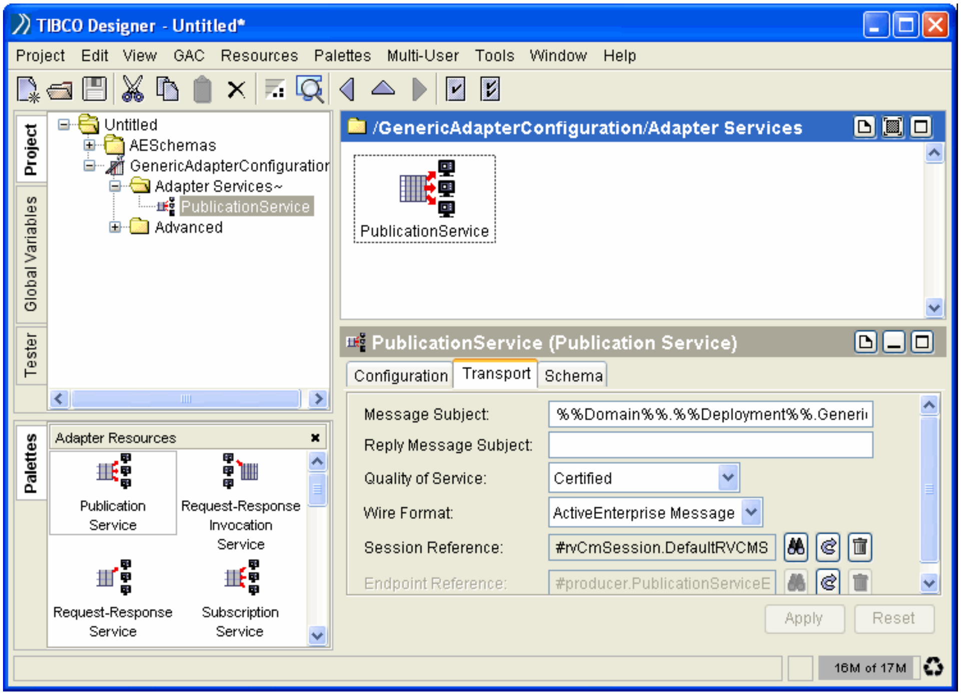
Here’s an overview of the steps involved in adapter configuration:

1. Using the TIBCO Designer adapter resource, specify the data the adapter publishes or subscribes to. You can specify multiple schema if different endpoints handle different data.
2. Drag a `Generic Adapter Configuration` into the design panel.

   TIBCO Designer automatically creates the appropriate base folders and objects.
3. In the Adapter Services folder, select the appropriate services (Publication Service, Subscription Service, and so on.) and drag them into the design panel, as shown in Figure 2.

A service is an abstraction that encapsulates an endpoint and the corresponding session.

Figure 2 Adapter Configuration in TIBCO Designer



Message Subject: %%Domain%%.%%Deployment%%.Generi

Reply Message Subject:

Quality of Service: Certified

Wire Format: ActiveEnterprise Message

Session Reference: #rvCmSession.DefaultRVCMS

Endpoint Reference: #producer.PublicationServiceE

4. For each service, specify the schema, the transport, and other attributes. When instantiating the service, TIBCO Designer automatically creates corresponding sessions and other objects and places them into the `Advanced` folder.

5. In certain situations, it may be necessary to customize the sessions themselves. You can do so by using the adapter's `Advanced` folder.

See Chapter 11, Custom Adapter Example: zapadapter for an example of a custom adapter configuration.

Exporting Project Repositories

After configuring an adapter in TIBCO Designer, export the adapter to a project repository. See the *TIBCO Designer User's Guide* for details.

Specifying Configuration Information

A custom adapter accesses the configuration information stored in the project repository. Separating code from configuration allows a custom adapter to run with one configuration during testing, and with a different configuration after deployment.

The location of the project repository can be specified on the command line, in a properties file, or in the code. In addition, some information about the adapter (for example, the instance ID or username and password) can be specified either in the custom adapter itself, on the command line, or in a properties file.

Command-Line Arguments

Command-line options have the highest precedence. A list of command-line arguments is given below.

```
-system:clientVar varName=value
-system:configurl relativeUrlPath | absoluteUrlPath
-system:propfile file
-system:repourl repositoryConnectionString | repositoryLocalFile
-system:instanceid instanceid
-system:plugin plugname
-system:version
-system:username username
-system:password password
-system:messageformat format
-system:jmsReconnectCount
-system:jmsReconnectDelay
-system:xsdGeneration
-system:dedicatedHawkThread
-system:showBanner
```

Table 13 gives detailed information of the command-line arguments.

Before using this feature, call `MAppProperties::setCommandLine()`.

Table 13 Command-Line Arguments

Argument	Description
`-system:clientVar` *<varName>=<value>*	Defines the value for a client variable in a repository. For example: `-system:clientVar cmName=foo` This value takes precedence over any global value set in the repository. Substitution takes place only at startup. If more than one `-system:clientVar` is specified in the command line, the latter one replaces the prior one. No space characters are allowed in either the `varName` or the `value` when using `-system:clientVar`. See Variable Substitution on page 34 for details. TIBCO Adapter SDK provides two predefined client variables: `AppName` - The name of the application set in `MAppProperties`. `InstanceId` - The name of the application instance.
`-system:configurl` *`<relativeUrlPath>`* | *`<absoluteUrlPath>`*	Specifies the location of the adapter instance description object to use. See Guidelines for Using the Argument –system:configurl on page 24 for details.
`-system:propfile` *<file>*	Tells the SDK to load a command properties file containing SDK startup information. It may also contain optional properties to be used by the adapter itself. These properties can be set and queried through the `MAppProperties::setProperty()` and `MAppProperties::getProperty()` methods (C++ only). For Java, the conventional properties mechanism can also be used. The properties file specification is part of the Java language. There are built-in APIs in Java for loading properties files, and the syntax of such files is detailed in the official Java documentation from Sun Microsystems. In particular, see the description of `java.util.Properties`. See Properties Files on page 39 for available keys and other information.
`-system:repourl` *`<repoConnectionString>`* | *`<repoLocalFile>`*	Specifies the location of the repository to use. See Guidelines for Using the Argument –system:repourl on page 25 for details.
`–system:instanceid` *<instanceId>*	`MApp InstanceId` to be used for this process. This information supersedes any other `instanceId` defined.
`-system:plugin` *`<plugname>`*	Loads a plug-in into the adapter. See the API documentation for the `MPlugin` class for more information.
`-system:`*<version>*	Gets the version of the application and SDK version the application is using.

Table 13 Command-Line Arguments (Cont'd)

Argument	Description
-system:username <*username*> -system:password <*password*>	The user name and password used by the repository server to access the security data.
-system:messageformat <*format*>	Specifies the message format to be used by the adapter. Legal values are `aeRvMsg`, `xmlRvMsg`, and `xmlJmsMsg`.
-system:xsdGeneration `on`|`off`	If this option is turned on, Adapter SDK validates an incoming XML message using the XSD for the given class. The default value is `off`. Refer to Working with XML and XSD on page 104 for more information.
-system:dedicatedHawkThread `on`|`off`	This property is specific to C++ SDK. By default, MApp runs as a single-threaded application. Under heavy load, this could interfere with Hawk's status update of the adapter application. Setting this property to on allows dedication of a separate thread for the application to do Hawk status updates. The default value is on.
-system:showBanner `true`|`false`	This property is specific to C++ SDK. Setting this property to `true` displays a banner containing application name, RepoURL, ConfigURL, application version, information string, and the TIBCO copyright notice. This banner appears on the console. The default value is `true`.
-system:jmsReconnectCount <*reconnection count*>	The JMS reconnection attempt count. For example: `-system:jmsreconnectcount=2`
-system:jmsReconnectDelay <*reconnection delay*>	The delay value between each JMS reconnection attempt. For example: `-system:jmsreconnectdelay=500`

Guidelines for Using the Argument –system:configurl

The argument `–system:configurl` *`<relativeUrlPath>`* | *`<absoluteUrlPath>`* specifies the location of the adapter instance description object. If not specified, the adapter instance description object provided in the `MApp` constructor is used.

- If *relativeUrlPath* is specified, the adapter instance description object is assumed to be under the default area in the repository (`/tibco/private/adapter/`).

  For example, the following command-line argument connects to an adapter instance description object named `ZAPInstance1` in the directory `/tibco/private/adapter/ZAPadapter`.

  ```
  –system:configurl ZAPadapter/ZAPInstance1
  ```

Here, `/tibco/private/adapter/` is the default area of the repository for adapter instance description objects, `ZAPadapter` is an application defined subdirectory.

- If *absoluteURlPath* is specified, the adapter instance description object is looked up in the repository.

  For example:

  ```
  -system:configurl
  /tibco/private/adapter/ZAPadapter/ZAPInstance1
  ```

Guidelines for Using the Argument –system:repourl

The argument `-system:repourl <`*`repoConnectionString`*`> | <`*`repoLocalFile`*`>` specifies the location of the repository.

- If *repositoryConnectionString* is specified, a connection is made to an adapter instance description object defined in a remote repository.

  The following example shows a connection to an adapter instance description object named `ZAPInstance1`, which is defined in a remote repository. No subject name is specified, so the default subject is used. Note that `tibrc@ syntax` is required as part of the connection string protocol.

  ```
  -system:repourl tibcr@ZAPInstance1
  ```

  The next example shows a connection string to an adapter instance description object defined in a remote repository that uses the given subject name for discovery. There are no space characters in the subject string.

  ```
  -system:repourl tibcr@ZAPInstance1:subject=foo.bar:
  service=7500:network=le0:daemon=tcp:target:7500:timeout=120
  ```

  In this example

  — *subject* is the subject for repository discovery.

  — *service*, *network*, and *daemon* are TIBCO Rendezvous parameters.

  — *timeout* is the timeout value in seconds for any TIBCO Administrator communication to be aborted.

- If *repositoryLocalFile* is specified, a connection is made to a local file repository.

  The following example connects to a repository instance named `ZAP.DAT` that is on the local file system.

  ```
  -system:repourl C:\repositories\ZAP.DAT
  ```

Property Key

Properties file values have the second highest precedence, which means if the same information can be set with a command-line argument, the command-line argument overrides the property key.

This section lists the property keys predefined and recognized by the Adapter SDK.

```
tibco.repourl
tibco.configurl
tibco.instanceid
tibco.appname
tibco.appinfo
tibco.appversion
tibco.username
tibco.password
tibco.clientVar.varname
tibco.messageFormat (values: aeRvMsg, xmlRvMsg, and xmlJmsMsg)
tibco.jmsReconnectCount
tibco.jmsReconnectDelay
tibco.xsdGeneration (values: off, and on)
tibco.dedicatedHawkThread (values: off, and on)
tibco.showBanner (values: true, and false)
tibco.jmsclientid.session_name
```

See Properties Files on page 39 for more information.

MAppProperties

The custom adapter code can set some of the configuration information, usually through the `MAppProperties` instance. This approach has the lowest precedence.

Specifying the information directly in the code, as shown in the following example, enables an adapter to always run with the same configuration:

```
MAppProperties appProperties;
appProperties.set(MAppProperties::APPNAME,"zapadapter");
appProperties.set(MAppProperties::APPVERSION,"3.0");
appProperties.set(MAppProperties::APPINFO,"Adapter SDK based ZAP
                  publisher" );
appProperties.set( MAppProperties::REPOURL,"tibcr@CPP_EXAMPLES");
appProperties.set( MAppProperties::CONFIGURL,
                  "examples/zapadapter/zapone");
appProperties.setCommandLine(argc,argv);
```

When the adapter calls the constructor for `MApp`, it passes in the `MAppProperties` instance.

Accessing Configuration Information

This section explains how the SDK encapsulates the information saved in the project repository, and how adapters access that information. It also includes a detailed description of the repository locator string.

Location of Configuration Information

The `repourl` and `configurl` specify the location of a project repository, which must be defined for a custom adapter.

- `repourl`: the repository location, that is, the location of the *project repository (repository instance)*.

  Specify the `repourl` in one of the following ways:

  — Calling the `MAppProperties` method `set` to set the `REPOURL` before creating the `MApp` application manager.

  ```
  appProperties.set(MAppProperties::REPOURL,"repoul")
  ```

  — Using the `-system:repourl` command line argument.

  — Setting `tibco.repourl` in a properties file.

- `configurl`: the location of the *adapter instance description object* inside the project repository.

  Specify the `configurl` in one of the following ways:

  — Calling the `MAppProperties` method `set` to set the `configurl` before creating the `MApp` application manager.

  ```
  appProperties.set(MAppProperties::CONFIGURL, "configurl")
  ```

  — Using the `-system:configurl` command line argument.

  — Setting `tibco.configurl` in a properties file.

See Server-based Repository Locator String on page 27 and Local Repository Locator String on page 30 for instructions on how to specify the locator string.

Server-based Repository Locator String

Applications built with the TIBCO Adapter SDK use a repository resource locator string to specify the server-based repository location and parameters.

The parameters available depend on the protocol with which the client and server communicate.

TIBCO Rendezvous

In the case of a TIBCO Rendezvous transport, a URL begins with `tibcr://` or `tibcr@`, followed by the instance name.

Table 14 lists the supported optional parameters. They are separated by colons.

Table 14 Optional Parameters for Server-based Locator String (TIBCO Rendezvous)

Parameter	Description
daemon	TIBCO Rendezvous rvd daemon value
service	TIBCO Rendezvous rvd service value
network	TIBCO Rendezvous rvd network value
rva	TIBCO Rendezvous rva host and port
subject	Instance discovery subject
discoveryTime	Timeout value in seconds for instance discovery
timeout	Timeout value in seconds for server requests
operationRetry	Number of retries when timeout occurs
userName	Any identifier (null or empty implies read only with guest privileges)
password	User password for security
regionalSubject	TIBCO Rendezvous subject prefix used for regional read-operation in the load balancing mode. For more information see the *TIBCO Administrator Server Configuration Guide.*
typeAccess	Type of client connection. Valid values are: `CLIENT_USAGE_DONT_CARE`—Client reads until update, then switches to write. This is the default. `CLIENT_USAGE_READ_ONLY`—Client is not allowed to do updates. `CLIENT_USAGE_READ_WRITE`—Client can do both reads and updates.
urlFile	Property file. The property file identifier can either be a fully qualified path or a relative path. The legal properties in this file are the same as optional parameters specified above. The properties in the file are appended to the repository locator string. If the same property appears in both locator string and property file, the properties in the locator string take precedence. Property values starting with # are considered obfuscated.

Examples are given below:

```
tibcr://myInst:service=5456:userName=ann:timeout=4000
tibcr@myInst:service=5456:urlFile=/tibco/props/fredsProps.txt
tibcr://myInst:urlFile=/tibco/props/fredsProps.txt
```

HTTP and HTTPS

In the case of HTTP transports, a URL begins with `http://`. In the case of HTTPS transports, a URL begins with `https://`.

The host name and port number come next (`http://host:port`). The port number is optional. If not specified, the default value is 8080 for HTTP and 8443 for HTTPS. In the case of HTTP/HTTPS clients, the host name and port number are followed by a '?' (`http://host:port`) and the instance name.

HTTPS-specific properties should be placed in a property file and that file should be specified using `urlFile=`. Therefore, `urlFile` is a required parameter for HTTPS.

In addition, remote HTTP/HTTPS clients support the following optional parameters separated by `&`. When `&` is used as separator of parameters and the URL is specified on the command line, the URL should be enclosed in quotes so that shell does not interpret it.

Table 15 Optional Parameters for Server-based Locator String (HTTP)

Parameter	Description
timeout	Timeout value in seconds for server requests.
operationRetry	Number of retries if a timeout occurs.
userName	Any identifier (null or empty implies read only with guest privileges)
password	User password for security
typeAccess	Whether it is read only or read-write. Valid values are: `CLIENT_USAGE_DONT_CARE`—Client reads until update, then switches to write. This is the default. `CLIENT_USAGE_READ_ONLY`—Client is not allowed to update. `CLIENT_USAGE_READ_WRITE`—Client can both read and update.

Table 15 Optional Parameters for Server-based Locator String (HTTP) (Cont'd)

Parameter	Description
urlFile	Property file. The property file identifier can either be a fully qualified path or a relative path. The legal properties in this file are the same as optional parameters specified above. The properties in the file are appended to the repository locator string. If the same property appears in both locator string and property file, the properties in the locator string take precedence. Property values starting with # are considered obfuscated.

Examples are given below:

```
http://host:8080/administrator/repo?myInst&userName=ann&timeout=40
00
https://host:8443/administrator/repo?myInst&urlFile=httpsProps.ini
https://host:8443/administrator/repo?urlFile=httpsProps.ini
```

Local Repository Locator String

Local repositories start with the instance name, which can optionally be preceded by `localrepo@`. The instance name can either be a fully qualified path or a relative path. The `.dat` extension is optional. In addition, clients support the following optional parameters separated by colons.

Table 16 Optional Parameters for Local Repository Locator String

Parameter	Description
userName	Any identifier (if not present or empty makes a read-only client)
urlFile	Property file. The property file identifier can either be a fully qualified path or a relative path. The legal properties in this file are the same as optional parameters specified above. The properties in the file are appended to the repository locator string. If the same property appears in both locator string and property file, the properties in the locator string take precedence. Property values starting with # are considered obfuscated.

Table 16 Optional Parameters for Local Repository Locator String (Cont'd)

Parameter	Description
typeAccess	Type of client connection. Valid values are: `CLIENT_USAGE_DONT_CARE`–Client reads until update, then switches to write. This is the default. `CLIENT_USAGE_READ_ONLY`—Client is not allowed to do updates. `CLIENT_USAGE_READ_WRITE`—Client can do both reads and updates. `CLIENT_USAGE_REACQUIRE_INSTANCE_LOCK`—Client is allowed to overwrite this local repository even if a lock file exists, as long as it's the same user. `CLIENT_USAGE_FORCE_INSTANCE_LOCK`—Client is allowed to overwrite this local repository even if a lock file exists. WARNING: Using this option may result in other users overwriting the project.

Examples are given below:

```
./instances/myInst.dat:userName=deborah
c:/tibco/repository/instances/myInst.dat:urlFile=c:/tibco/reposito
ry/props/deborah
myProj.dat
myProj
myProj/myrepo.dat
```

How Adapters Access Configuration Information

This section explains how a custom adapter can access configuration information.

MApp and MClassRegistry

Upon startup, `MApp` creates an `MClassRegistry` object that the custom adapter can use to access the information in the project repository. For example:

- Use transport configuration information to create one or more transport sessions.
- Create endpoint objects. Endpoints are publisher, subscriber, client, or server.
- Perform automatic creation of timers.
- Create reporting sinks, as specified in TIBCO Designer for that adapter instance, and map them to the trace object created by `MApp`.
- Create the Metadata description classes specified by the schema configuration data. See Guidelines for Metadata Use on page 102.

In addition, `MApp` creates metadata description classes for metadata objects and their attributes. The custom adapter can access that information as needed. See Metadata Description Classes on page 96.

MProperties

By default, `MApp` creates an `MProperties` instance that encapsulates access to the adapter instance description object.

In the following C++ code fragment, the adapter creates a publisher using the configuration URL and the subject specified in the adapter instance definition. Note how the configuration URL is included as the first argument to `getValue()`.

```
if (!pMProperties->getValue("zapadapter/zapconnection/instance", m_sInstanceId))
      m_sInstanceId = "zap:112";

   if (!pMProperties->getValue("zapadapter/zapconnection/hostname", m_sHostName))
      m_sHostName = "hostile";

   if (!pMProperties->getValue("zapadapter/zapconnection/port", m_iPort))
      m_iPort = 8000;
```

A Java SDK custom adapter uses similar methods.

Configuration Classes

A custom adapter uses configuration classes to access configuration information. The following classes are defined in both the C++ and the Java API:

- `MProperties`—Custom adapters call methods in an instance of this class to retrieve property information.
- `MPropertiesEnumerator`—Allows you to iterate through the `MProperties` name/value pairs.
- `MPropertiesRegistry`—Manages all `MProperties` instances. Allows you to commit and roll back changes and create new `MProperties` instances for project repositories.
- `MConfigurationUtilities`—Allows specialized custom adapters direct control over the creation of components and metadata from configuration information. This class should only be used under certain circumstances.

The Java SDK has the following classes for encapsulating information retrieved from an `MProperties` instance: `MPropertyElement`, `MPropertyAttribute`, and `MPropertyText`. This helps custom adapters retrieve information from `MProperties`.

See `MProperties.elements()` in the *TIBCO Adapter SDK API Reference* for more information.

MProperties.elements() Example

The following example shows how to enumerate an `MProperties` object.

```
/* get the configuration properties from MApp */
MProperties prop = mapp.getConfigProperties();
Enumeration enum = prop.elements("/private/tibco/adapter");
for (;enum.hasMoreElements();) {
           Object o = e.nextElement();
           if( o instanceof MPropertyAttribute)
           {
               MPropertyAttribute a = (MPropertyAttribute)o;
              System.out.print(" "+a.getName()+"="+a.getValue() );
           }
           else if(o instanceof MPropertyText)
           {
               MPropertyText t = (MPropertyText )o;
               System.out.println( t.getText() );
           }
           else ifo instanceof MPropertyElement)
           {
               MPropertyElement t = (MPropertyElement)o;
                   System.out.println
                     "This is PropertyElement - has to recursively
                      get the enumeration again");
           }
           else
               System.out.println("Should never happen");
}
```

Variable Substitution

The Adapter SDK variable substitution mechanism can override global variables predefined in the project repository in a restricted manner. Predefined variables can be viewed and set in TIBCO Designer. Variables are specified as `%%VARNAME%%` and contain no white space.

Variable Substitution Mechanism

Variable substitution allows users to accomplish the following tasks:

- Substitute string variables specified in the repository at startup time.
- Locally define the value for a variable in the properties file for a specific project repository. The local value takes precedence over any global value.
- Specify the value for a variable through a command-line argument. This overrides all other specification.

  `-system:clientVar` *varName=value*

  Multiple variables can be specified using the command line `-system:clientVar`. If the same variable already exists in the command line, the newly input value replaces the existing value. No space characters are allowed for either *varName* or the *value*, when using `-system:clientVar`.
- Specify the value for a variable in a properties file. This overrides the project repository and values set in code, but not variables set on the command line.
- Enforce the predefined variables listed in Predefined Global Variables on page 36.

  Variables can be used anywhere in the configuration and will be replaced by the locally defined adapter instance.

Specifying Variables

Custom adapters can specify variables in the following ways:

- In the project repository during configuration:
  - using TIBCO Designer (recommended)
  - editing repository XML files (not recommended)
- In a properties file (using the `tibco.clientVar.<`*`varname`*`>` property)
- On the command line (using `-system:clientVar` *varName=value)*

The order of precedence is: command-line values overwrite values set in the properties file, properties file values overwrite values set in the repository.

Specifying Variables Using TIBCO Designer

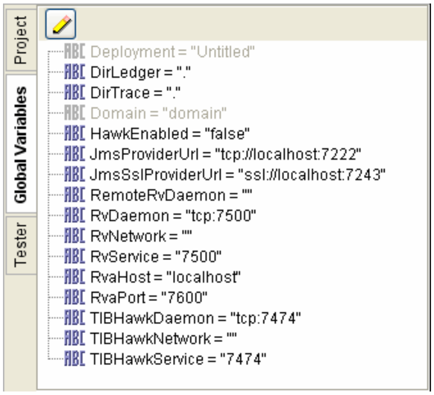
Global variables provide an easy way to set defaults for use throughout a project.

For example, assign the value 7474 to the predefined global variable `RvDaemon`, then use the variable in different sessions in an adapter. If you want to change the TIBCO Rendezvous daemon for an adapter, globally set it to a different value or override it from the command line.

To specify global variables using TIBCO Designer:

1. Start TIBCO Designer.
2. In the project panel, click the **Global Variables** tab. All currently defined global variables are displayed in the project panel.

Figure 3 Global Variables Tab



Deployment = "Untitled"
DirLedger = "."
DirTrace = "."
Domain = "domain"
HawkEnabled = "false"
JmsProviderUrl = "tcp://localhost:7222"
JmsSslProviderUrl = "ssl://localhost:7243"
RemoteRvDaemon = ""
RvDaemon = "tcp:7500"
RvNetwork = ""
RvService = "7500"
RvaHost = "localhost"
RvaPort = "7600"
TIBHawkDaemon = "tcp:7474"
TIBHawkNetwork = ""
TIBHawkService = "7474"

3. To edit the global variables, click the **Open Advanced Editor** button ✎ .

   In the Global Variables dialog, you can do the following:

   — To assign or change a variable value, click the field in the Value column and then enter the new value. Press **Enter** when done.

   — To add a global variable group, click the **Add a Variable Group** button ▦. Specify the name of the group, then press **Enter**.

   — To add a global variable, click the **Add a Variable** button abc. A new global variable item is added to the bottom of the list. With a variable group selected, you can click this button to add variables to the group. Enter the variable name and, optionally, the value. Press **Enter** when done.

— To add a global variable to a group, select the desired group icon and click the **Add a Variable** button.

4. To use the global variable in the fields of a resource, enter the variable name enclosed with `%%` on both sides. For example, `%%DirTrace%%`.

   When the project is deployed and the configured components are run, all occurrences of the global variable name are replaced with the global variable value (unless it was overridden in a way that had higher precedence).

A number of global variables are predefined. See Predefined Global Variables on page 36. You can add definitions of any variables you need to the predefined variables.

Specifying Variables in the Properties File

To specify variables in a properties file, use the following parameter:

`tibco.clientVar.`*varname*

Specifying Variables on the Command Line

To specify one or more variables on the command line, use the following syntax:

`-system:clientVar` *`varName=value`*

Predefined Global Variables

Table 17 lists the predefined global variables. Some global variables are automatically used within the system when an adapter instance is configured.

Table 17 Predefined Global Variables

Variable	Description
`Deployment`	Defaults to the TIBCO Designer project name. It can be any string value. This global variable is used by the system to partially define the subject name defined for a service.
`DirLedger`	The path name of the TIBCO Rendezvous certified messaging ledger file. The default is the root installation directory.
`DirTrace`	The path name for log file used by the adapter. The default is the root installation directory.

Table 17 Predefined Global Variables (Cont'd)

Variable	Description
`Domain`	The default value for a file-based local project is `MyDomain`. The value for a server-based project is the domain to which the project was saved.
`HawkEnabled`	Indicates whether TIBCO Hawk is used to monitor the adapter. `True` indicates that a Hawk microagent is defined for the adapter. `False` indicates the microagent is not to be used. Default is `False` in TIBCO Designer but `True` for SDK-based adapters.
`JmsProviderUrl`	Tells applications where the JMS daemon is located. Setting this value mostly makes sense in early stages of a project, when only one JMS daemon is used.
`RemoteRvDaemon`	TIBCO Rendezvous routing daemon (`rvrd`) to be used. See *TIBCO Administrator Server Configuration Guide* for instructions on how to set up a domain using `rvrd`.
`RuntimeCertificatesDirectory`	Provides support for external SSL files at runtime by allowing applications to reference external trusted certificates instead of using the certificate in the EAR file. Use this variable to specify the path of the directory in which all the Trusted Certificates are stored. This variable is valid only for TIBCO ActiveMatrix BusinessWorks and TIBCO JMS. For more information on SSL, refer to *TIBCO Designer User's Guide*.
`RuntimeRvDaemonCertificate`	Provides support for external SSL files at runtime by allowing applications to reference external trusted certificates instead of using the certificate in the EAR file. Use this variable to specify the path of the certificate file when the supported transport is TIBCO Rendezvous.
`RvDaemon`	TIBCO Rendezvous daemon. Sessions use this daemon to establish communication. The default value is `7500`.
`RvNetwork`	TIBCO Rendezvous network. This variable needs only be set on computers with more than one network interface. If specified, the TIBCO Rendezvous daemon uses that network for all outbound messages. In most cases, you can leave the default.

Table 17 Predefined Global Variables (Cont'd)

Variable	Description
`RvService`	TIBCO Rendezvous service. The Rendezvous daemon divides the network into logical partitions. Each transport communicates on a single service. A transport can communicate only on the same service with other transports. Unless you are using a non-default TIBCO Rendezvous configuration, leave the default (7500).
`RvaHost`	Computer on which the TIBCO Rendezvous agent runs. This variable is only relevant if you are using the TIBCO Rendezvous Agent (`rva`) instead of the TIBCO Rendezvous daemon, and if you have configured a non-default setup. See *TIBCO Rendezvous Administration* for instructions on how to specify the rva parameters.
`RvaPort`	TCP port where the TIBCO Rendezvous agent (rva) listens for client connection requests. See *TIBCO Rendezvous Administration* for instructions on how to specify the rva parameters. Defaults to 7501.
`TIBHawkDaemon`	TIBCO Rendezvous daemon used in the TIBCO Hawk session. See the *TIBCO Hawk Installation and Configuration* manual for details.
`TIBHawkNetwork`	TIBCO Rendezvous network used by the TIBCO Hawk session. See the *TIBCO Hawk Installation and Configuration* manual for details.
`TIBHawkService`	TIBCO Rendezvous service used by the TIBCO Hawk session. See the *TIBCO Hawk Installation and Configuration* manual for details.

Properties Files

The Adapter SDK supports properties files, which contain property key/property value pairs that would otherwise be supplied on the command line. Properties files are supported with both the C++ and the Java API.

If an adapter includes a properties file, the precedence of options is affected.

- The properties file overrides everything except those options entered from the command line.
- Options directly entered on the command line override everything else.

An option "`-system:propfile` *file*" is *not* supported within the properties file itself. Therefore, properties files cannot be nested. You may, however, call an SDK properties file from a TIBCO Wrapper properties file. See Chapter 13, TIBCO Wrapper Utility.

Format of Properties File

Each line in a properties file is a single property. Each property consists of a key and a value. The key starts with the first non-whitespace character and ends at the first "=", "`:`", or whitespace character. The value starts at the first character after the equal sign (=).

The properties file format is identical to the Java properties file format defined in `java.util.Properties.load()`. Therefore, the following restrictions apply:

- The "`!`" character cannot be used as a comment line indicator. Only the "`#`" character is recognized.
- The line continuation character is ignored (no multi-line values).
- The key cannot contain any of the termination characters. Although Java allows termination characters by escaping the value with a preceding "\" character, TIBCO Adapter SDK does *not* support this syntax.

Tagging Values for Encryption

The presence of "`#!`" as the first two characters in a value (not the key) indicates that the value has been encrypted or is to be encrypted.

The encryption tool can be found in `tra/`*version*`/bin`. It is also called `obfuscate` for compatibility. When the encryption tool is run, it rewrites the properties file with the encrypted value in place.

Obsolete

Using "#" for obfuscation is obsolete.

Recognized Property Keys

The following property keys are predefined and recognized by the Adapter SDK:

```
tibco.repourl
tibco.configurl
tibco.instanceid
tibco.appname
tibco.appinfo
tibco.appversion
tibco.username
tibco.password
tibco.clientVar.varname
tibco.messageFormat (values: aeRvMsg, xmlRvMsg, and xmlJmsMsg)
tibco.jmsReconnectCount
tibco.jmsReconnectDelay
tibco.xsdGeneration (values: off, and on)
tibco.dedicatedHawkThread (values: off, and on)
tibco.showBanner (values: true, and false)
tibco.jmsclientid.session_name
```

Two Types of Properties

The Adapter SDK allows two types of properties in a properties file:

- *SDK properties* affect the SDK program. They are listed in Recognized Property Keys on page 40.
- *TIBCO Wrapper properties* affect the TIBCO Wrapper. The TIBCO Wrapper utility is separate from the SDK libraries and must be explicitly linked in if you want to use it. See Chapter 13, TIBCO Wrapper Utility for more information.

Both types of properties can be included in one file. Alternatively, you can place each in an individual file and call the SDK properties file from the TIBCO Wrapper properties file using `application.args --propfile` in the latter.

Properties File Examples

Example Properties File

```
tibco.configurl=/tibco/private/adapter/test/config/config1
tibco.repourl=tibcr://SDK_TEST
tibco.appname=MyApp
tibco.instanceid=config
tibco.appversion=4.0
tibco.appinfo=TIB/Adapter SDK test for Configuration
tibco.username=admin
tibco.password=samplePassword
tibco.clientVar.service=7600
tibco.clientVar.daemon=tcp:7600
tibco.showBanner=false
```

Setting the Properties File Inside the Program

Custom adapters that want to set the properties file in code instead of passing it on the command line can refer to the following example.

```
main()
{
   MAppProperties myAppProps;
   //…
   myAppProps.set( MAppProperties::PROPFILE, "obfuscatedFile" );
   //…
   MString r3password;
   if ( myAppProps.get( "r3.password", r3password )== Mtrue )
   {
      // log into R/3…
   };
//…
}
```

Chapter 3 Adapter Program Elements

This chapter discusses the main elements of an adapter program.

Topics

Overview

The primary task of an adapter is to retrieve or send data. The SDK supports this with a flexible architecture that allows separation of the configuration from the adapter program. The SDK contains the facility to work in this framework and the classes to encapsulate the transport, endpoint, and so on.

The program elements of an adapter fit together as follows:

- The `MApp` application manager handles initialization and shutdown of an adapter. Applications can customize initialization and shutdown. See MApp Application Manager on page 45.
- The adapter interacts with the source or target application using either a publish/subscribe or a request/reply transport protocol. See Transport Protocol on page 51.
- Endpoints send or receive the data. They are the publishers, subscribers, clients, and servers in a custom adapter. See Endpoints on page 53.

  Endpoints are configured to use a specific message format, which packages the data going over the network. See Transports, Wire Formats, and Message Formats on page 56.

  Each endpoint is associated with a session. The `MSession` subclasses encapsulate the transport (TIBCO Rendezvous or TIBCO Enterprise Message Service) used to communicate with the source or target application. See Sessions on page 58.
- The program is executed based on an event model using event sources (which are the endpoints that receive or send data), events, and event listeners. See Event Model on page 60.

MApp Application Manager

At the center of each running adapter instance is the application manager, an instance of a subclass of `MApp`. All adapters need to create a subclass of `MApp` and define its methods.

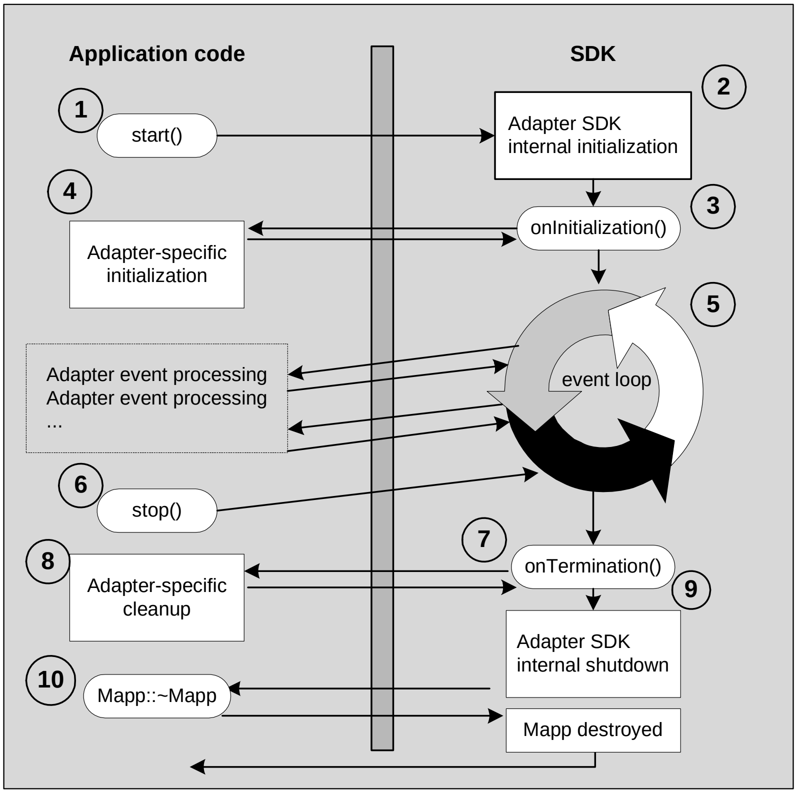
The `MApp` instance handles the following tasks:

- Initialization and shutdown of an adapter. Loading the configuration data (such as sessions, endpoints) and metadata into memory as part of initialization.
- Connecting to the project repository and processing configuration and metadata information found there.
- Setting up tracing and tracking and TIBCO Hawk management.
- Providing the glue logic that binds components together.

The SDK `MApp` class and its application-defined subclass implement these tasks together.

Top-Level Control Flow

Figure 4 Adapter Control Flow (C++)



Control Flow in C++

Figure 4 illustrates how a C++ program proceeds in single-threaded mode. In general, custom adapters run most efficiently in single-threaded mode, which is also the default mode. If a custom adapter requires multithreading, see Multithreaded Adapters on page 124.

The default steps are as follows:

1. The custom adapter creates an instance of a subclass of `MApp` and calls its `start()` method.

2. When `MApp` receives the call to `start()`, the SDK performs internal initialization:
   a. Checks to see if `MApp` is already started; if it is, just returns.
   b. Reads the information in project repository and creates objects as appropriate. This includes session, endpoint, and trace objects, as well as metadata description objects.
   c. Creates the default TIBCO Hawk microagent.
   d. Activates all sink components.
3. `MApp` calls the method `onInitialization()`, which performs application-specific initialization and must be defined for each custom adapter.
4. The SDK executes `onInitialization()`.
5. When `onInitialization()` returns successfully, `MApp` behaves as follows:
   - If `start()` was called with `bStartEventLoop` set to `true` (default behavior), `MApp` enters a loop to dispatch events.
   - If `start()` was called with `bStartEventLoop` set to `false`, `MApp` returns execution control to the custom adapter, which can then call `MApp::nextEvent()` to dispatch any incoming event, or `MRvSession::nextEvent()` to dispatch events for this session.

   See Event Model on page 60 for information on the event model in use.
6. To exit the event loop, the custom adapter must send a message that invokes the `stop()` method, which leads to a shutdown operation.

Also, call `stop()` if the program never entered the event loop.

7. When `MApp` receives the call to `stop()`, it calls `onTermination()`—defined by the custom adapter—which performs application-specific cleanup.
8. The custom adapter executes `onTermination()` and returns control to the SDK.
9. The SDK performs system-specific shutdown and returns control to the adapter.
10. The adapter calls the destructor for `MApp`.
11. `MApp` is destroyed and control returns once more to the adapter.

Control Flow in Java

The control flow for Java differs slightly from that in C++. The SDK and the custom adapter interact as follows:

1. The custom adapter creates an instance of a subclass of `MApp` and calls its `start()` method.
2. When `MApp` receives the call to `start()`, it performs internal initialization:
   - — Checks to see if `MApp` is already started; if it is, just returns.
   - — Reads the information in project repository and creates objects as appropriate. This includes session, endpoint, and trace objects, as well as metadata description objects.
   - — Creates the default TIBCO Hawk microagent.
   - — Activates all sink components.
3. After `MApp` has performed internal initialization, it calls `onInitialization()`. This method performs application-specific initialization and must be defined for each custom adapter.

   Components and metadata objects are already available; the custom adapter can therefore register listeners with subscribers and activate the subscribers.
4. The SDK executes `onInitialization()`, then activates all components, if so specified. For some components, additional criteria must be met. For example, a subscriber must have at least one listener attached to get activated.
5. The SDK spawns a new thread in which the event manager runs, and returns control to the custom adapter.

   See Event Model on page 60 for information on the event model in use.
6. When the custom adapter is ready to exit the event loop, it sends a message to invoke the `stop()` method.
7. When `MApp` receives the call to `stop()`, it first checks whether `MApp` is already stopped; if so, it just returns. If not, it calls `onTermination()`.
8. The adapter executes `onTermination()` and returns control to the SDK.
9. The SDK performs a system-specific shutdown that:
   - — Stops the event thread.
   - — Deactivates all components.
   - — Performs other internal cleanup.

Creating an MApp Instance

Custom adapters set up `MApp` as follows:

1. Use the TIBCO Designer software to set the adapter name, version, banner information, and other information, and save the configuration to the project repository.
2. Programmatically create an `MAppProperties` instance that points to the project repository.

   See `MAppProperties` in the online API Reference documentation.
3. Call the constructor for `MApp`, passing in the `MAppProperties` instance, which allows access to the configuration information.

`MApp` is an abstract class. All custom adapters must create a subclass and implement, at a minimum, the `onInitialization()` and `onTermination()` methods. In addition, `MApp` offers methods to allow command-line arguments, to retrieve the current adapter name and version, and so on. See the online API documentation for `MApp` for details.

Examples

Example 1 onInitialization()

The following code fragment illustrates the use of `onInitialization()`.

```
class ExampleApp extends MApp
{

   ExampleListener listener;
   public ExampleApp(MAppProperties appProperties )
   {
      super(appProperties );
   }

   protected void onInitialization() throws MException
   {
      // attach listener for a subscriber for discovery messages
      MComponentRegistry componentRegistry =
      getComponentRegistry();
      MSubscriber sub = componentRegistry.getSubscriber(
      "DiscoveryListener" );
      sub.addListener( listener );

      // attach to a monitored object to a Hawk microagent
      MHawkRegistry hawkRegistry = getHawkRegistry();
      Object monitoredStats = new Stats( this );
      MHawkMicroAgent hma = hawkRegistry.getHawkMicroAgent(
      "GetStatistics" );
      hma.setMonitoredObject(monitoredStats );
   }
…
}
```

Example 2 onTermination()

The following code fragment illustrates the use of `onTermination()`.

```
public void onTermination() throws MException
{
   // application specific database cleanup
   dbConn.commit();
   dbConn.close();

   // global notification that application is stopping
   MPublisher publisher = getComponentRegistry().getPublisher(
   "Alert-sender" );
   Minstance message =
   getClassRegistry().getDataFactory().newInstance( “Alert” );
   message.set( "status", "Stopping" );
   publisher.send(message );
}
```

Transport Protocol

TIBCO Adapter SDK allows custom adapters to use a publish/subscribe or a request/reply transport protocol. This section gives an overview of both protocols. See Chapter 7, TIBCO ActiveEnterprise Operation Model, on page 113 for more information about request/reply (client/server) interactions.

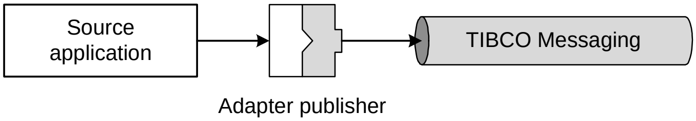
Publish/Subscribe Protocol

Publish/subscribe interactions are driven by events such as the arrival of data or a timer signaling that a specified interval has elapsed.

Adapter Publisher

A publisher gets information from a source application, for example, an ERP (Enterprise Resource Planning) application, and makes it available (publishes it) to the transport, directly or through an intermediary.

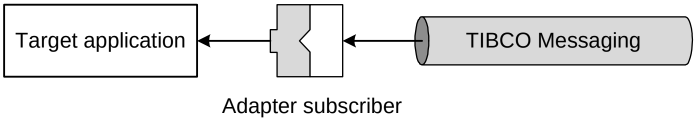
Figure 5 Adapter Publisher



Adapter Subscriber

A subscriber receives information from the transport and passes it on to a target application.

Figure 6 Adapter Subscriber

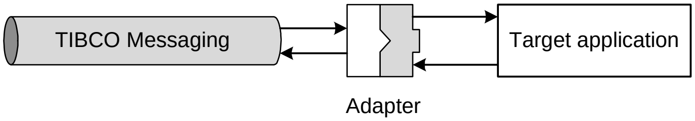


In publish/subscribe interactions, data producers (publishers) are decoupled from data consumers (subscribers), that is, they do not coordinate data transmission with each other. Information is routed to the appropriate consumer because that consumer has subscribed to all messages with the matching destination.

Request/Reply Interactions

Demand for data drives request/reply (client/server) interactions. A client requests data from a server; the server computes an individual response and returns it to the client. Communication flows in both directions, as illustrated in Figure 7. The complete interaction consists of two point-to-point messages—a request and a reply.

Figure 7 Demand-Driven Request/Reply Interactions



Request/Reply Basics

Demand-driven computing is well-suited for distributed applications that require point-to-point messages. In request/reply interactions, data producers coordinate closely with data consumers. A producer does not send data until a consumer makes a request.

The server sends replies to the client that requested the data. The client listens until it receives the reply, and then stops listening (unless it expects further installments of information).

Request/reply interactions can be implemented in two ways:

- `MPublisher` and `MSubscriber` have facilities to implement request/reply.
- ActiveEnterprise operations support a more sophisticated approach to request/reply interactions. See Chapter 7, TIBCO ActiveEnterprise Operation Model, on page 113.

Request/Reply and Reply Destinations

To perform request/reply interactions and set your own reply destination, do the following:

1. Call the `MPublisher` method `setReplyDestination()` to specify the reply subject.
2. Create the `MSubscriber`(s) listening on the reply subject.
3. Call the `MPublisher` method `send()` to send the message.

Note that the `MPublisher` method `sendWithReply()` is not suited for this approach because it is always point-to-point.

Endpoints

Endpoints represent the services that an adapter provides. Publisher, subscriber, client, or server instances are the endpoints in a custom adapter.

TIBCO Designer allows to create endpoints by using the concept of a service. A service encapsulates both an endpoint and the corresponding session. Services include:

- Publication Service—a publisher and associated session
- Subscription Service—a subscriber and associated session
- Request-Response Service—a server and associated session
- Request-Response Invocation Service—a client and associated session

Services are abstractions used for GUI configuration purposes. The SDK itself encapsulates endpoints and sessions but not services.

Creating Endpoints

Endpoints can be configured through TIBCO Designer or be created programmatically using the appropriate constructor. It is recommended that you use TIBCO Designer for configuration because it enables changing endpoint details (for example, transport information) without recoding.

Configuring Endpoints in TIBCO Designer

To configure an endpoint and use it in a custom adapter:

1. Open TIBCO Designer and create a project.
2. Select the `Generic Adapter Configuration` resource and drag it into the Design panel.
3. Select the `Generic Adapter Configuration` instance in the Project panel, click its `Adapter Services` folder to open it.
4. Select one of the resources (for example, Publication Service) available in the Palette panel and drag it into the Design panel.
5. Specify the relevant information of the service. For example: name, transport type (Rendezvous or JMS), information about the transport in the Transport tab (including wire format and quality of service), schema describing the data the endpoint sends or receives.
6. Save the project.

Creating Endpoints Programmatically

To create endpoints programmatically:

1. Instantiate an `MRvEndpointSpec` or `MJmsEndpointSpec`.
2. Assign member variables to the endpoint. These variables specify, among other things, the message format for the endpoint to use.
3. Pass the endpoint specification to the `MPublisher` or `MSubscriber`.

Because of the advantages of separating configuration from the code, it is preferable to include endpoints in the configuration instead of creating them programmatically.

Changing Endpoint Quality of Service

The SDK-based adapters can be written to work with different types of sessions. For example, suppose you have configured an adapter for TIBCO Rendezvous Certified Messaging and have written the corresponding adapter code. If you later decide to switch to TIBCO Enterprise Message Service with explicit confirmation, the code will still work (unless you performed some unusual customizations).

In contrast, if you have written code that expects the endpoints to use automatic confirmation, and then change the configuration of the endpoints to use a non-persistent transport, the sending adapter continues to wait for the receiving adapter to confirm the events. As a result, resources will be consumed without being freed.

Assume a publisher has been configured to use RVCMQ. In that case, the following code fragment would send messages in `aeRvmsg` wire format.

```
MClassRegistry * pCR;
 MInstance outInstance( pCR , "sdkSchema" );

 //Set the instance ..
 outInstance.set( ... , ... );
 outInstance.set( ... , ... );
 ...
 ...

 //Serialize into Mtree
 MTree rTree;
 outInstance.serialize( rTree );

 //Publish the message
 pMPublisher->send( rTree );
```

Suppose you want to change the message to be sent in using TIBCO Enterprise Message Service and, because of that, `aeXml` wire format. You can change the endpoint configuration using TIBCO Designer and the code will work, but there will be serious performance degradation.

The instance is first converted to `MTree` using `aeRvmsg` wire format. When the `MTree` is sent out by the JMS endpoint, the `MTree` is deserialized back to `Minstance` and then converted to the an `MTree` using `aeXml` wire format. The code should therefore be changed as follows:

```
MClassRegistry * pCR;
 MInstance outInstance( pCR , "sdkSchema" );

 //Set the instance ..
 outInstance.set( ... , ... );
 outInstance.set( ... , ... );
 ...
 ...

 //Publish the message
 pMPublisher->send( outInstance );
```

The `MInstance` is now sent directly, and the SDK takes care of the serialization.

Change all the programs to send out `MInstance` objects instead of serializing the `MInstance` to `MTree`.

Transports, Wire Formats, and Message Formats

This section discusses transports, wire formats, and message formats. Understanding what each term means, and what the available options are, is essential for proper configuration and programming of the adapter.

Adapter events are created by an event source such as an `MTimer` or an `MSubscriber` (see Event Model on page 60). With each event, data are either sent or received using one of the supported transports and wire formats. You must specify the transport, quality of service, and wire format when configuring the adapter elements.

Transports

Both TIBCO Rendezvous and TIBCO Enterprise Message Service are supported. Each transport supports different qualities of service:

- TIBCO Rendezvous supports reliable, certified, and transactional messages. In addition, you can use TIBCO Rendezvous to support queues.
- TIBCO Enterprise Message Service supports persistent or non-persistent delivery modes. Standard JMS topics and queues are supported.

SDK C++ API calls the EMS C API to create lookup context and other EMS functionalities as there is no concept of `tibjmsnaming` in the EMS C API. The concept of `tibjmsnaming` is used only in Java for JNDI implementation.

To use `tibjmsnaming`, specify the TCP notion in Designer or SDK, and specify `tibjmsnaming` in the `factories.conf` file, which is queried by the EMS server.

In the `factories.conf` file, specify the URL of `[TopicConnectionFactory]` or `[QueueConnectionFactory]` as follows:

```
[TopicConnectionFactory]
type = topic
url = tibjmsnaming://hostname:7222
```

Wire Formats

Three wire formats are supported:

- `rvMsg` (standard TIBCO Rendezvous Message)
- `aeRvMsg` (ActiveEnterprise Messages, which includes control information)
- `aeXml` (standard XML representation)

TIBCO Rendezvous supports all three wire formats, while TIBCO Enterprise Message Service supports only `aeXml`. See TIBCO Adapter Wire Formats on page 136 for more information.

Message Formats

The combination of a transport and a wire format is called a message format. A message format is a C++ enumerated type or a Java interface that can be passed, for example, when constructing an `MTree` class. The following message formats are available.

Table 18 Message Formats

C++	Java	Description
`M_RV_MESSAGE_FORMAT`	`MMessageFormat.RV`	`rvMsg` using TIBCO Rendezvous transport
`M_AERV_MESSAGE_FORMAT`	`MMessageFormat.AERV`	`aeRvMsg` using TIBCO Rendezvous transport
`M_XMLRV_MESSAGE_FORMAT`	`MMessageFormat.XMLRV`	`aeXml` message using TIBCO Rendezvous transport
`M_XMLJMS_MESSAGE_FORMAT`	`MMessageFormat.XMLJMS`	`aeXml` message using JMS transport

Sessions

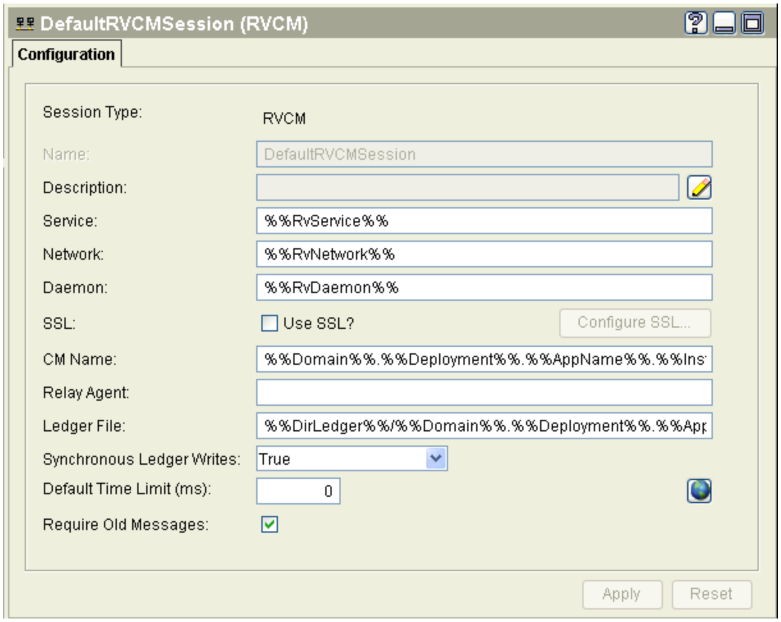
An SDK session encapsulates transport information. To represent sessions the SDK supplies a hierarchy of classes: a parent class, `MSession`, and two subclasses, `MRvSession` and `MJmsSession`.

Each adapter instance can use one or more sessions. The SDK session configurations can be created and changed through TIBCO Designer. This keeps configuration separate from the running application.

Sessions can also be defined programmatically. However, using TIBCO Designer is preferred for flexibility.

When creating adapter services in TIBCO Designer, a corresponding endpoint (publisher, subscriber, etc) and session are created automatically. Figure 8 shows a default RVCM Session created when a Publication Service was instantiated. In general, you do not need to change these defaults.

Figure 8 Default Session



DefaultRVCMSession (RVCM)

Configuration

Session Type: RVCM

Name: DefaultRVCMSession

Description:

Service: %%RvService%%

Network: %%RvNetwork%%

Daemon: %%RvDaemon%%

SSL: ☐ Use SSL?

CM Name: %%Domain%%.%%Deployment%%.%%AppName%%.%%Ins

Relay Agent:

Ledger File: %%DirLedger%%/%%Domain%%.%%Deployment%%.%%Apr

Synchronous Ledger Writes: True

Default Time Limit (ms): 0

Require Old Messages: ☑

If multiple sessions are required for the same endpoint, select the session in the configuration's `Advanced` folder in TIBCO Designer. When you save the project, the information is stored in the project repository.

In C++, timers are associated with a TIBCO Rendezvous session. In the Java SDK, they are session-independent. (There is no such thing as a JMS-based timer.)

In Java, TIBCO Rendezvous sessions can be RVA sessions. RVA sessions allow communication with the RVA daemon instead of the RVD daemon. Note that unlike RVD, the RVA daemon is not started automatically.

Sessions and TIBCO Rendezvous

When you create services and allow TIBCO Designer to auto create a session, you need not to worry about the type of session. If, however, you create an `MPublisher` or `MSubscriber` that uses TIBCO Rendezvous programmatically, ensure that the appropriate TIBCO Rendezvous session (transport) is available.

Table 19 Endpoints and TIBCO Rendezvous Sessions

Protocol Used by Endpoint	Required Session
RV	Any type of session: RVA (Java only), RV, RVCM, or RVCMQ (subscriber only).
RVCM	RVCM. If the session is of type RV, an error occurs.
RVCMQ (subscriber)	RVCMQ. If the session is of type RV, an error occurs.

Multiple Sessions

SDK-based adapters can use multiple sessions. By default, sessions are autocreated by TIBCO Designer when creating a Publication or Subscription Service.

You can also use the `Advanced` folder inside the adapter configuration in TIBCO Designer to define as many sessions of each type as you need. During execution of `MApp::start()`, the SDK will create the corresponding instances of the appropriate `MSession` subclass. Custom adapters can obtain information about a session by calling the methods in the session instance.

Event Model

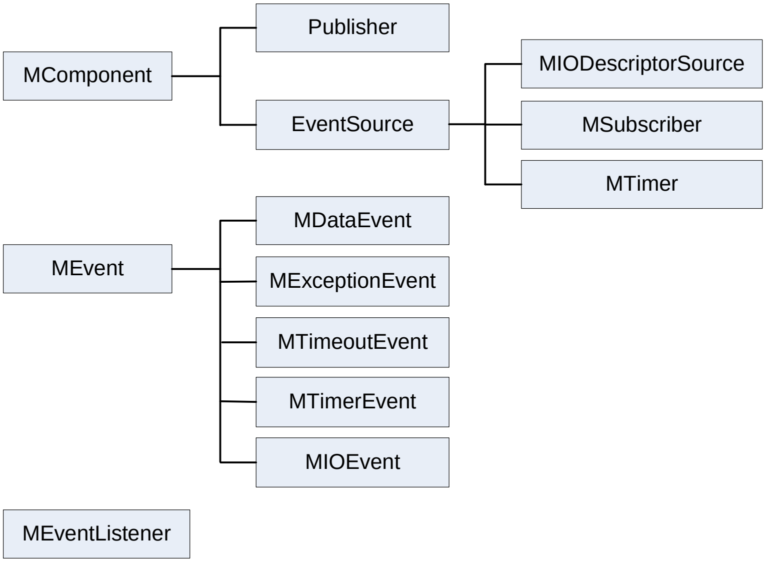
This section discusses the TIBCO Adapter event model.

- Event Management Classes, page 60
- TIBCO Adapter Flow of Event Information, page 61
- Extending Adapter Event Classes, page 62

Event Management Classes

Three classes participate in the TIBCO Adapter event model: event sources, events, and event listeners. Figure 9 illustrates the event management class hierarchy.

Figure 9 Event Management Class Hierarchy



- **Event sources** generate events, that is, an event source creates an event instance when it is triggered. An example is the `MSubscriber` event source, which creates `MDataEvent`, `MExceptionEvent`, or `MTimeOutEvent` instances.
- **Events** are created and owned by the event source. Instances of events encapsulate state information about an event. Some adapters may decide to provide a custom event for their source or target application.

An `MSubscriber` sends a special event, `MExceptionEvent`. See How Adapters Receive Data on page 69 for details on when a subscriber receives an `MException` event.

- **Event listeners** register interest with event sources. Custom adapters create appropriate subclasses of the SDK event listener class and implement `onEvent()` methods with application-specific behavior.

The following code fragment accesses a subscriber event source, which was defined in the project repository, and binds a listener—in effect, a callback—to the subscriber:

```
MSubscriber* pMSubscriber  = pComponentRegistry ->
getComponentByName ("mySub");

DataEventHandler* pDataEventHandler = new DataEventHandler( this );
pMSubscriber->addListener( pDataEventHandler );
```

The Java API supports `MEventSource`, `MEvent`, and `MEventListener` classes to implement an event model that mirrors that of the C++ API.

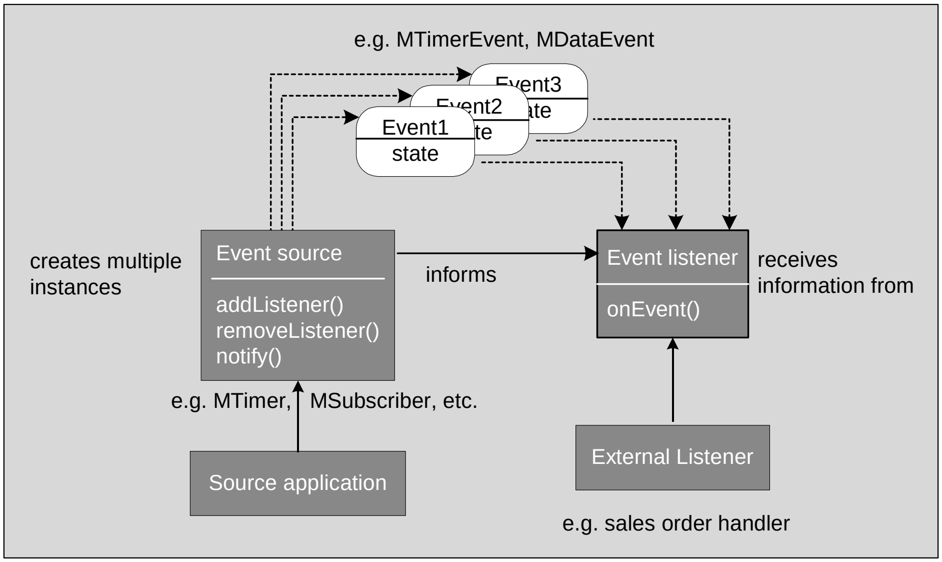
In contrast to the C++ API, the `MIODescriptor` event source and related `MIOEvent` are not supported because there is no need for them in conjunction with the Java event model.

TIBCO Adapter Flow of Event Information

Assume the custom adapter has been set up to listen to certain data events. In that case:

1. An external process (source application) publishes data.
2. An `MSubscriber` instance (event source) receives the data and creates an instance of `MDataEvent` (event). A new event instance is created each time new data is received.
3. The event source informs each registered listener by calling the listener's `onEvent()` method. Each `onEvent()` method executes using the information stored in the `MDataEvent` instance.

Figure 10 TIBCO Adapter Event Model



Extending Adapter Event Classes

A custom adapter can use custom events to suit its needs. In these cases, the custom adapter needs to provide the following elements:

- Custom event source(s)—subclass(es) of `MEventSource`—with these methods defined:
  - `addListener()` adds a listener to the event source.
  - `removeListener()` removes a listener from the event source. Calling this method *does not* cancel certified delivery agreement for RVCM.
  - `notify()` triggers an event in each event listener attached to this event source.

  Custom adapters can also remove listeners from the event source.
- A custom event (subclass of `MEvent`). This is the event the custom event source generates. In the constructor for the event, the related event source is provided.
- Instances of `MEventListener` with appropriate `onEvent()` methods.

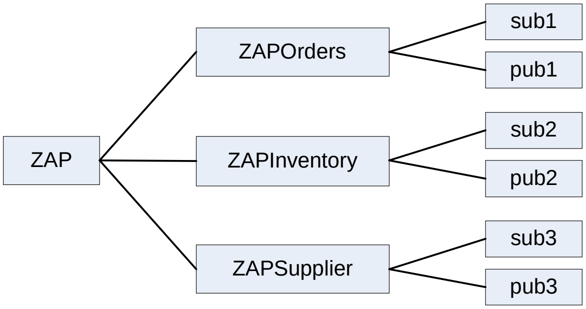
Multiple Adapter Instances

You can run more than one instance of the same adapter on one or multiple computers. The instances can use the same configuration data or different configuration data. Adapter configurations are defined using TIBCO Designer. Each adapter instance can have different sessions, endpoints, timers, and metadata.

Figure 11 shows the ZAP adapter with several instance definitions. Each instance definition starts as a clone of the ZAP adapter and has a unique set of configuration data defined.

The project repository for each adapter is in a separate user-defined adapter directory, and uses a unique instance ID and project name. The information for each project is then used by the adapter program at runtime.

Figure 11 One Adapter with Multiple Instances



Chapter 4 Sending and Receiving Data

This chapter discusses how adapters send and receive data.

Topics

Adapter Application Data

An adapter works with two kinds of data, application data and metadata. This chapter discusses application data. Metadata are discussed in Chapter 6, Metadata, on page 89.

Application Data Overview

Adapters need an in-memory data structure to represent complex, hierarchical information that can be sent over a network. There are two options:

- `MData` subclasses (`MInstance, MSequence, MUnion`) are restricted by metadata information.
- `MTree` is a generic structure for complex hierarchical data. There is no constraint of content based on metadata.

MInstance, MSequence, MUnion

The Adapter SDK provides several subclasses of `MData` to represent complex hierarchical information:

- `MInstance`. The `MInstance` class allows hierarchical data representation based on a predefined class description (metadata information). This metadata information is provided in the repository. The `MApp` application manager encapsulates the metadata information so it can be used during `MInstance` creation.
- `MSequence`. Class for encapsulating lists of data.
- `MUnion`. Class for encapsulating data specified in the repository as a union. Unions have a name and contain one or more `unionMember` association lists. Each `unionMember` association list has a name and a class. Union and union member elements may have extended properties.

In this manual, `MInstance` refers to one of these composite objects. In most cases, custom adapters send `MInstance` objects.

In SDK 5.x and later, an `MInstance` can be published without additional modification. The SDK serializes the `MInstance` when you pass it to a publisher's `send()` method.

MTree

In addition to the `MData` subclasses, which are always constrained by metadata, SDK also supports an `MTree` format.

An `MTree` is a data structure that can be sent over a network. When sending an `MData` subclass to the transport, the SDK serializes it to create an `MTree`. When the data arrives, it is in `MTree` format.

Application Data Message Format

When a publisher endpoint sends data over the wire, the data must be marshalled to include message format information.

When sending an `MInstance`, the SDK performs the serialization and you need not be concerned about the message format. See Sending MInstance Data on page 68.

The message format contains information about the transport (TIBCO Rendezvous or TIBCO Enterprise Message Service) and the wire format (aeRvMsg, rvMsg, aeXml). The following message formats are supported.

Table 20 Message Formats

C++	Java	Description
`M_RV_MESSAGE_FORMAT`	`MMessageFormat.RV`	`rvMsg` using TIBCO Rendezvous transport
`M_AERV_MESSAGE_FORMAT`	`MMessageFormat.AERV`	`aeRvMsg` using TIBCO Rendezvous transport
`M_XMLRV_MESSAGE_FORMAT`	`MMessageFormat.XMLRV`	`aeXml` message using TIBCO Rendezvous transport
`M_XMLJMS_MESSAGE_FORMAT`	`MMessageFormat.XMLJMS`	`aeXml` message using JMS transport

How Adapters Send Application Data

This section discusses sending data using the SDK `send()` method. For simplicity, the discussion uses a publisher as an example. The same concepts apply if data is sent as part of a client/server interaction.

Sending MInstance Data

An SDK `MPublisher` has various `send()` methods that take an `MInstance` (or other composite `MData` subclass) as an argument. Because the publisher has been created to use one of the supported message formats, you can pass in the `MInstance` and the publisher handles the serialization appropriately.

Sending MTree Data

Custom adapters can also send an `MTree`. In that case, the custom adapter must explicitly serialize the `MInstance`.

It is recommended that you use the `MPublisher.send()` method to send an `MInstance` and let the SDK perform serialization.

If you explicitly serialize the `MInstance`, the serialization method must know which message format to use. Choose one of the following options:

- Set a global variable (`MESSAGEFORMAT)` to determine the message format. This variable is then used by the SDK each time the `MTree` is serialized. Set the global variable as follows:`MAppProperties.set(MESSAGEFORMAT, ...)`
- Pass the message format as an argument, so the SDK can use the correct format for serialization. Here is a sample code fragment:

  ```
  MInstance        foo( ... );
  // set the data into the foo instance
  //...
  MTree        x;
  foo.serialize( x, Pub.getMessageFormat() );
  Pub.send( x );
  ```
- Accept the default, `aeRvMsg`.

If an `MTree` is produced with a message format that does not match the sending endpoint, the endpoint is forced to convert the `MTree` to the matching message format. The performance cost of doing so is often unacceptable for production systems.

After you have serialized the `MInstance`, call the `send()` method.

How Adapters Receive Data

This section discusses receiving data. For simplicity, the discussion uses a subscriber as an example. The same concepts apply if data is received as part of a client/server interaction.

When custom adapters receive data, the data always arrives in `MTree` format. The message format of the subscriber determines how data is handled. The following rules apply:

- If the incoming message format matches the message format with which the subscriber was configured, an instance of the `MDataEvent` results. The custom adapter can then deserialize the `MTree` that contains the data, and use the resulting `MInstance`.

  For example, the custom adapter application code could enter the data into a database or enterprise application, depending on the task the adapter has been set up to perform.

- If the protocol used by the publisher differs from that of the subscriber, the subscriber never gets the message.
- If the protocol is the same, but the wire format is not matched, the subscriber *usually* sends an `MExceptionEvent` that includes the message data. That way, the receiving custom adapter knows that something unusual has happened but still gets the data.
- If the protocol is the same, and the wire format expected by the subscriber is `rvMsg`, the subscriber passes on the data that arrived, regardless of the content of the message, in an `MDataEvent`.

Chapter 5 Tracing, Tracking, and Exception Handling

TIBCO Adapter SDK provides a number of ways to send error or debugging information. This chapter discusses tracing, tracking, and exception handling.

Handling TIBCO Rendezvous advisories is discussed in Chapter 8, Advanced Features, page 123.

Topics

Tracing

Traces are useful both during development (debugging messages) and after the custom adapter has been deployed (error and information messages).

The SDK allows custom adapters to store error message information in a separate file. In the file, each message includes a number of elements, such as the message code, the message role and category, and the message symbol. The custom adapter program uses the message symbol, and the SDK maps the symbol to the message in the file to send out a complete message to the desired location. Storing error messages separate from the actual custom adapter program makes it easier to localize the adapter to a specific language locale.

This section discusses the following topics:

- Using Tracing Facilities, page 72
- Available Tracing Roles, page 74
- Multiple Traces and Sinks, page 74
- MTrace and MSink, page 75
- Sample XML Message Element, page 76
- Trace Message Format, page 77
- Configuring Tracing Using TIBCO Designer, page 78
- File Sink Management, page 79

Using Tracing Facilities

This section gives an overview of how to implement tracing in a custom adapter. Some of the example programs included in the TIBCO Adapter SDK (for example, zapadapter) use this approach.

1. In TIBCO Designer, specify the sinks you want to have available and their associated roles. See the *TIBCO Designer Palette Reference* for more information.
2. Prepare an XML file containing elements for each message you expect to use. See Sample XML Message Element on page 76. Include a role for each message element. You can use one of the predefined roles or a user-defined role. See Available Tracing Roles on page 74.
3. Process the XML file using the `genAeErrors` utility (in the *SDK_HOME*/`resourceKit` directory).

The utility can have two possible outputs:

- — Source files that can then be included in the project. This can be either a `.cpp` and a `.h` file or a `.java` file. The output files contain an `MMessageBundle` and an `InitializeMessages` definition.
- — If you are using Java, you may choose to output a Java properties file. In that case, load the properties file as follows:

  ```
  MMessageBundle.addResourceBundle("ZAP","ZAP_Messages");
  ```

See the example program for details.

The output file includes a message symbol, which is then to be used in your code. When later call `trace()` with the message symbol as an argument, all other message elements are used in the trace.

4. Be sure to call the `InitializeMessages()` static method in the `onInitialization()` method in `MApp`.
5. During initialization, `MApp` creates an instance of `MTrace` that you can retrieve as follows:

C++

```
MTrace *pMTrace = pMapp->getTrace();
```

Java

```
MTrace trace = mapp.getTrace();
```

6. In the custom adapter, call the `MTrace` method `trace()` with an `MException` or an error code. For example:

C++

```
//.....
pTrace->trace( "ZAP_ERR_NO_CONNECTION", NULL, NULL );
```

Java

```
try {
//...
} catch (MException ex) {
    MTrace.trace("ZAP_ERR_NO_CONNECTION", ex, null);
}.
```

It is recommended that you use the `genAeErrors` utility and the `InitializeMessages` method instead of creating the `MMessageBundle` explicitly and adding messages. See the API documentation for more information.

7. At runtime, the SDK looks up the message corresponding to the message symbol used in the `trace()` call and sends the complete trace message to the appropriate sink.

See the *TIBCO Adapter SDK Satus Codes* for a complete list of predefined messages and codes. You may create additional XML files with message codes and related information for the custom adapter.

Available Tracing Roles

The SDK predefines the roles listed in Table 21. In addition, user-defined roles are supported. For instructions on creating user-defined roles, refer to the *TIBCO Designer Palette Reference*.

Table 21 Predefined Roles

C++	Java	Description
`MAPP_DEBUG_ROLE`	`MTrace.DEBUG`	Developer-defined tracing. During normal operating conditions, DEBUG should be turned off.
`MAPP_ERROR_ROLE`	`MTrace.ERROR`	Unrecoverable errors. The operation in which the error occurred is skipped. The adapter may continue with the next operation or may stop altogether.
`MAPP_INFO_ROLE`	`MTrace.INFO`	A significant processing step was reached. It is logged for tracking and auditing purposes.
`MAPP_WARNING_ROLE`	`MTrace.WARN`	An abnormal condition was found. It does not prevent processing to be performed but special attention from an administrator is recommended.

Multiple Traces and Sinks

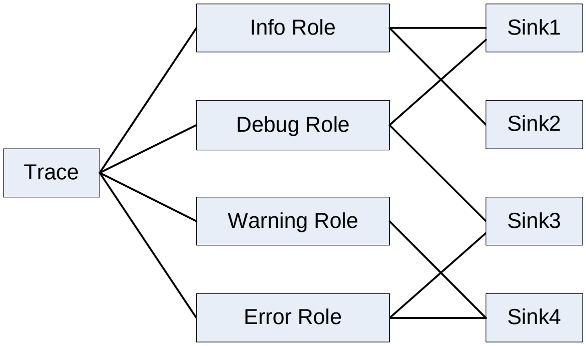
With multiple roles and sinks, the output information level can be tuned via modifications to the adapter instance definition. Because each sink receives trace messages corresponding to its assigned role, you can do the following:

- Send different types of information (info, debug, warning, error) to different sinks.
- Send the same information to more than one sink. For example, send it to a file and publish it as a TIBCO Rendezvous message.

  Combinations of sinks and roles are specified using TIBCO Designer.

In Figure 12, all error trace messages go to Sink4 and Sink3; all warning messages go to Sink4; all debug messages go to Sink1 and Sink3; and all info trace messages go to Sink1 and Sink2.

Figure 12 Assigning Different Trace Messages to Different Sinks



MTrace and MSink

The two SDK classes `MTrace` and `MSink` offer the following services:

- Send a message to the desired location. The following sink types are supported:
  - — `MFileSink` Messages are sent to file.
  - — `MRvSink` Messages are sent to TIBCO Rendezvous.
  - — `MStdioSink` Messages are sent to a user-provided file; usually `stderr` or `stdout`.
  - — `MHawkSink` Messages are sent to TIBCO Hawk, to the specified microagent method `_onUnsolicitedMsg()`.
- Format a message string (`printf()` style).
- Fine-tune where and when different types of information are sent.

The SDK implements trace message handling as follows:

1. The developer uses the TIBCO Designer software to define sinks and map each sink to one or more roles.

2. When a custom adapter calls the `MApp` method `start()`, `MApp` prepares for tracing as follows:
    1. Creates an `MTrace` instance.
    2. Attaches the `MTrace` to the `MApp` from which it was called.
    3. Creates the sinks defined for the adapter instance.
    4. Maps trace roles and sinks. Roles group trace messages so related messages can then be examined in one place. There are system-defined roles (error, information, etc.) or you can use a user-defined role.
3. Each time the custom adapter calls one of the `MTrace` tracing methods, a role is implicitly specified via the error code that is passed in.
4. Each sink that has been mapped to the specified role receives that message if the role has been turned on.

See File Sink Management on page 79 for information on naming and potential deletion of log files.

Sample XML Message Element

The following message element illustrates how to specify messages inside the trace message XML file.

The XML file is compiled into source files or a Java property file when you run the `genAeErrors` utility. At runtime, the compiled message is accessed to map from the message code to other message information so the whole trace message can be displayed, sent to the log, etc.

```
<message>
        <messageCode>AESDKC-0041</messageCode>
        <messageSymbol>M_ERR_CANNOT_FIND_PREFIX</messageSymbol>
        <role>errorRole</role>
        <category>SerializerCategory:
        <description>Deserialization failed to find prefix in Idx list
            </description>
        <longDescription/>
        <resolution>Deserialization was attempted on an MTree that does
         not contain necessary metadata prefix list information. Either correct
         the sending application or do not attempt to deserialize this MTree (it
         may not have been packed by an SDK-based application)</resolution>
</message>
```

Trace Message Format

In preparation for running a custom adapter, prepare an XML file that matches error codes with other error information. See Sample XML Message Element on page 76. Then launch the `genAeErrors` tool, which generates sources or resource bundles from the XML file.

The message is typically extracted from the `MMessageBundle` object at runtime. A substitution might occur in a reference field using `Error` description to fill the field. This ensures each error definition originates from one, and only one, source.

In the log file, each message has the same format, derived from the tracing facility. Table 22 lists each field and corresponding examples.

Table 22 Trace Message Fields

Field	Example
<*timestamp*>	2000 Sep 12 18:13:26:373
<*applicationId*>	`zap.inst1` (The default `applicationId` format is <*appName.instanceID*>)
<*role*>	Error
<*category*>	[TibrvComm]
<*message code*>	AESDKJ-0237
<*message*>	AE Operation Request timeout
<*tracking information*>	tracking= #<trackingID>#<info1>#<info2>#<info3>#

Here is an example of a single trace message:

```
2000 Sep 12 18:13:26:373 zap.inst1 Error [TibrvComm] AESDKJ-000237
AE Operation Request timeout tracking=
#<trackingID>#<info1>#<info2>#<info3>#
```

Table 23 gives details for each field in the trace message.

Table 23 Trace Message Details

Field Name	Required	Description
Timestamp	Yes	Timestamp of occurrence.
ApplicationId	Yes	Name of the product where the error is raised. A combination of the SDK application name and `instanceID` is appropriate and used by default.

Table 23 Trace Message Details (Cont'd)

Field Name	Required	Description
Role	Yes	Role of the trace message. See Available Tracing Roles on page 74 for a definition of the following predefined roles: `ERROR`, `WARN`, `INFO`, and `DEBUG`.
Category	Yes Not required for DEBUG messages.	Category of the message. **Standard categories**: `Application`, `Component`, `Configuration`, `Database`, `JMSComm`, `Marshaller`, `Metadata`, `Properties`, `Rpc`, `Serializer`, `TibRvComm`, `Unicode`, `Utility`, `XML` If necessary, the standard category set can be extended but extension is discouraged
MessageCode	Yes Not required for DEBUG messages.	Unique code for the message. The code consists of one alphanumeric key followed by a dash "-" and a four-digit code.
Message	Yes	Text message. Only string substitutions are acceptable.
TrackingId	No	Always a GUID defined by the source. Set by the source and carried forward by intermediate components. Note that in all data-related messages the `trackingId` should be traced.
ApplicationInfo	No	Application-specific information added to the tracking info to trace the message back to its source. Set initially by the source and carried forward, it is augmented by each intermediate component. For adapters, `ApplicationInfo` should be in the following form: `<appName>.<instanceID>.<businessEvent>.<BusinessId>`

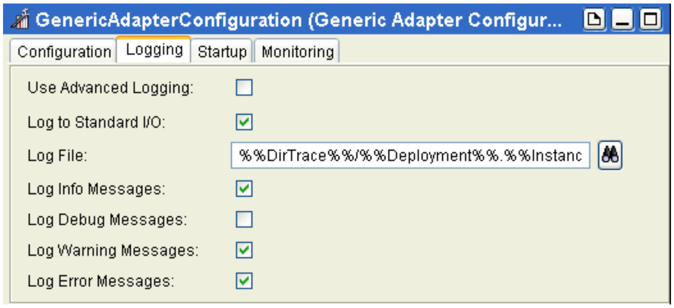
Configuring Tracing Using TIBCO Designer

This section demonstrates how to configure tracing using TIBCO Designer.

By default:

- the Generic Adapter Configuration is set up to support tracing to a log file named according to a variable;
- Info, Warning, and Error messages are being logged;
- Debug messages are not logged.

Figure 13 Configure Tracing



Use Advanced Logging: ☐

Log to Standard I/O: ☑

Log File: %%DirTrace%%/%%Deployment%%.%%Instanc

Log Info Messages: ☑

Log Debug Messages: ☐

Log Warning Messages: ☑

Log Error Messages: ☑

You can change the default tracing by selecting or clearing the check boxes in the Logging tab of the Generic Adapter Configuration. You can also change the name of the log file.

For any other changes, for example, for additional sinks, select the **Use Advanced Logging** check box.

Advanced Logging

With the Use Advanced Logging check box selected, you can create additional sinks or change the default sinks from inside the Advanced folder's `Log Sinks` folder.

See the *TIBCO Designer Palette Reference* for more information.

File Sink Management

In most cases, custom adapters specify a file sink in the adapter configuration, which includes `File Limit` and `File Count` attributes. The SDK uses this information as follows:

1. Creates a file using the file name assigned for the adapter instance.
2. Writes to that file until it reaches the size specified in `File Limit`.
3. When `File Limit` is reached, the SDK renames the current file to *file*.1 and creates a new file with no extension.

   Note that the log file may be slightly larger than the limit because the new file is only created *after* the limit has been reached.

4. The SDK repeats this process, renaming all files each time a new file is generated, until it reaches the number of files specified in File Count.

The adapter overwrites the file with the lowest number, that is, the oldest file, when the number of files reaches `File Count` and that last file reaches `File Limit`. To avoid that, set either `File Count` or `File Limit` to a sufficiently large value.

Setting File Sink Information

To set non-default File Sink information in TIBCO Designer:

1. In the Generic Adapter Configuration's Logging tab, click **Use Advanced Logging**.
2. In the Advanced folder for the adapter, drag a **File Sink** into the design panel.
3. Specify the information for the file sink.

Tracking

The TIBCO Adapter SDK tracking facility allows custom adapters to associate tracking information with data (*not* with trace messages). The tracking information remains associated with the data as they are passed between ActiveEnterprise applications. This makes it possible to see a history of the operations performed on the message.

Use of tracking should not be confused with tracing. While tracing provides full detail of custom adapter activities, tracking provides an audit trail for messages as they travel through the ActiveEnterprise.

MTrackingInfo

Tracking is implemented by an `MTrackingInfo` object.

1. The source application places information into the `MTrackingInfo` objects using the `MTrackingInfo::addApplicationInfo()` method.

   This SDK adds the information to the `MTree`, `MData`, or `MOperation`.
2. As the data is sent through the enterprise, each TIBCO application adds information about its operations to messages that pass through.
3. If an operation on the data fails, the application where it fails can access the information by looking at the message.

`MTrackingInfo` cannot be used if you are using `rvMsg` format.

Because the `MTrackingInfo` is sent on the wire, applications should limit the information added through `addApplicationInfo()`. Each application should call `addApplicationInfo()` only once per message. The message should contain concise and minimal information.

Tracking Example

The following pseudo-code fragment illustrates how a subscriber event listener could extract tracking information if serialization failed.

```
// *** A Subscriber tracking info sample ***
void
MyMEventListener::onEvent( MEvent const & event )
{
    MTree const * pTree = NULL;
    MDataEvent const * pDataEvent = MDataEvent::downCast( &event );
```

```
        if ( pDataEvent ) {
            pTree = pDataEvent->getData();
            try {
                MInstance foo( m_pClassRegistry, *pTree );
                //...
            }
            catch ( MException e ) {
                MTrackingInfo info = pTree->getTrackingInfo();
                m_pMTrace->trace( e, &info, NULL );
            }
            //...
        }
    }
```

Exception Handling

The TIBCO Adapter SDK has its own `MException` class.

Exceptions in the C++ API

The C++ `MException` class differs from the standard C++ exception classes in that it does not use a different exception subclass for each type of exception. Instead, `MException` encapsulates exception information, including an exception code, to avoid a large number of exception classes in the SDK. See the *TIBCO Adapter SDK Message Codes* for more information.

Catching an `MException` instance is similar to catching any other C++ exception. `MException` accessors allow the custom adapters to access the encapsulated information. Comparison and assignment operations are also available. It is possible to get a verbose description of the exception. See `MException` in the C++ API.

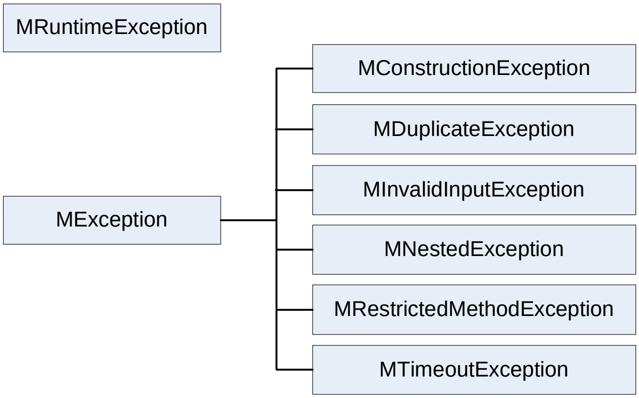
An `MOperationException` class has been defined for use with the SDK operation model, which allows custom adapters to perform operation invocation or to act as an operation server in a client-server implementation. Custom adapters do not create instances of `MOperationException`, they only retrieve error information from it. See TIBCO ActiveEnterprise Operation Model on page 113.

Exceptions in the Java API

The SDK supports standard Java exception handling. The Java API includes an `MException` class with subclasses that indicate different types of exceptions. Each class is documented as part of the API documentation.

Java custom adapters can use the `instanceof` operator to find out the specific error that occurred. If the custom adapter wants to print the error to stdout, it can call the `getMessage()` method from the exception object to retrieve the message string.

Figure 14 SDK MException Class Hierarchy



Using Exceptions

A custom adapter can catch exceptions thrown by the SDK and handle them appropriately. Custom adapters can also create their own subclass of `MException` for application-specific exceptions.

```
try {
...
catch (MException e){
m_pTrace ->trace (e);
...
```

Designing an Exception-Handling Mechanism

When designing an exception-handling mechanism, consider the following recommendations:

- Distinguish between business and system errors.
  - — When an operation fails due to business logic execution, it should not prevent the adapter from continuing to function normally. The adapter must anticipate and handle business exceptions. You should also consider the possibility of receiving data that does not match the expected metadata description.
  - — For programming mistakes, the adapter must ensure that there is enough information traced and logged for an administrator to debug or work around the problem.

- Handle business errors inside or outside the custom adapter.
  - Business errors can be handled outside the adapter (for example, in TIBCO ActiveMatrix BusinessWorks).
  - A business exception handling mechanism within the adapter is also possible. However, such design could compromise the flexibility of the adapter to evolve over time as business requirements change.
- Consider using TIBCO Hawk or TIBCO Administrator monitoring to help manage system errors.
  - Hawk can help monitor and manage the adapter using a Hawk trace sink at runtime. Hawk can also be used to restart an adapter in case of fatal system errors. Hawk sends out alert message, email, or a page to an administrator who can correct and bring the system back up if needed.
  - TIBCO Administrator implements monitoring and tracing behavior. You can also write trace information to files.
- Consider whether transactional behavior applies for the adapter.
  - Transactional messaging behavior provided by TIBCO Rendezvous TX includes atomic transaction, exactly once message consumption (duplicate detection), and store and forward message delivery.
  - You can commit on success and handle transactional exceptions with rollback.
- Maintain transactional integrity when handling system errors.
  - You should ensure that the adapter can survive system errors such as a target application down. Message integrity must be preserved in case of system errors.
  - By wrapping a transaction around both the consumption of the incoming message and the publication of the resulting transformation, the engine is made lossless. That is, a message is never marked as consumed until the transformed version has been successfully published. Similarly, no transformed message is published until the incoming source message has been marked as consumed.
- Consider disaster scenarios and disaster recovery schemes.
  - All known system and business exceptions should be handled. Logging and monitoring can help manage other disaster scenarios.

Exception Handling in Delayed Acknowledgement of Certified Messages

Delayed acknowledgement of a certified message is also known as explicit RVCM confirmation. This feature is most useful when controlling whether to confirm receipt of the certified message. This feature also allows you to handle the case where a target application crashes after it receives a certified message, but before it can process the message.

Note that successful certified message delivery is different from successful application processing of the received message. Adapter business logic and application error (not application crash type) handling logic should be separated from explicit RVCM confirmation.

Adapter developers must decide whether the adapter should acknowledge messages that have resulted in an error.

First, a developer should differentiate between data errors and system errors:

- **Data errors** If the message is retransmitted, it will probably result in the same error. This may not always be true if the operation failed due to missing information that may subsequently have become available (for example, a foreign key violation).
- **System errors** The message cannot be processed because the target system is down. The message is retransmitted after the system comes back up and the message may then be successfully processed.

Second, being a message delivery protocol, RVCM has no built-in intelligence to retransmit on a failure condition at application level. RVCM should not be used to recover or retransmit after an error condition, such as failed operation invocation.

Certified messages are only retransmitted under the following conditions:

- When the receiving certified subscriber detects a missed message from the sequence number. For example, the receiver gets message # 1,2, 3 then 5. The certified subscriber will request retransmission for message # 4.
- In recovery from abnormal termination of either the publisher or subscriber. For example, a certified subscriber crashes while the certified publisher keeps publishing. When the subscriber is restarted, the messages published while it was down will be retransmitted from the publisher.

Messages that result in data errors should be acknowledged (confirmed) and logged, which allows the problem to be addressed outside the adapter.

Messages that result in system errors should be handled in one of the following ways:

- Explicit RVCM confirmation can be used and the adapter can simply not acknowledge a failed message. However, the adapter must manage pausing and triggering retransmission of the message so that it can be resent for processing.

The Adapter SDK provides a subscriber method that allows custom adapters to suspend and restart certified message delivery subscription. The `MSubscriber::suspend()` method stops the current subscription interest but does not cancel the certified delivery agreement. The `suspend()` method works the same way as an abnormal termination of subscriber, mentioned above. By suspending subscription interest the publisher retains all the certified messages it publishes.

When the target application is reconnected the subscriber can resume subscription and the publisher will retransmit all messages retained. Subscription interest is restarted by calling the `MSubscriber::activate()` method. There are no equivalent methods for the request response service, you should instead use the operation remote exception handling mechanism (that is, confirm and raise remote exception).

- If the subscriber application is a stateless, short-lived process, sending the RVCM confirmation at the end of the process is acceptable. Otherwise you need to persist and confirm, then do the processing. If a system error occurs you should recover using the persistence.

Either way, you must deal with the possibility of a continuous system error (that is, the target system is never reconnected) and should handle it like any data error.

Chapter 6 Metadata

The chapter explains how to use TIBCO Adapter SDK metadata.

Topics

Understanding TIBCO Adapter Metadata Management

The SDK metadata classes form the basis for model-driven computing with an adapter. Metadata is, by definition, data about data. Metadata isolates the data description from the application data itself.

In regards to the SDK, metadata is data describing the data the custom adapter deals with. For example, for a custom adapter that extracts data from a database and publishes it to TIBCO Rendezvous, the metadata would describe the database data to be extracted.

Metadata is also known as schema data. The two terms can be used interchangeably.

If several applications handle the same data, a specification document must exist that defines exactly how the data is implemented. Each time the specification document changes, all applications dependent on it must change as well.

Instead of a specification document, developers can use metadata to describe the application data. The metadata is used in a similar fashion to a hardcopy document, however, applications can access metadata automatically and don’t have to manually update data descriptions.

Well-designed metadata makes it possible to have an adapter with the same behavior under different circumstances. If data definitions in the source or target application change, you change the metadata that describes the data instead of changing the custom adapter code and recompiling. In addition, standardizing on metadata provides interoperability among TIBCO ActiveEnterprise components.

Uses for Metadata

Custom adapter developers specify the metadata hierarchy in TIBCO Designer and save them as part of the project repository. Metadata is useful in several ways:

- Metadata describe the data model (schema) for the custom adapter. When the data model of the source or target application changes, you do not have to rewrite the adapter. Instead, you change the metadata objects.
- By changing the metadata you can change which data the custom adapter publishes or subscribes to. It is therefore possible to use the same adapter code with different sets of metadata residing in different directories in different project repositories.

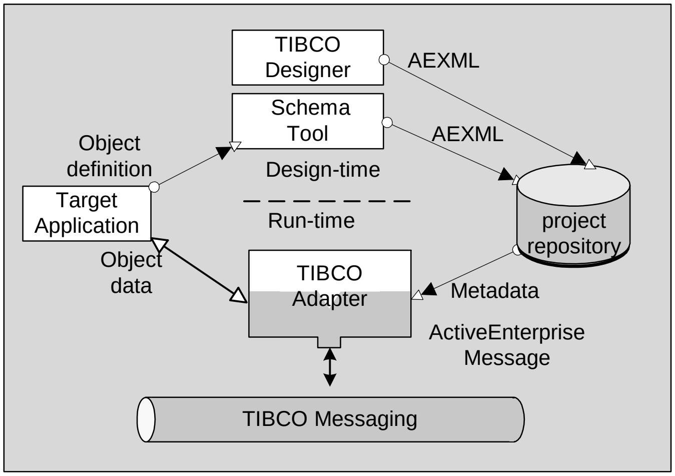
- Metadata encapsulates the data themselves as classes and the relationships between classes as associations. Much of the information in the source data can therefore be preserved.

Custom adapters create instances of the `MInstance` class based on metadata information. The SDK serializes the `MInstance` objects when they are send out so they become instances of `MTree`. On the wire, data is always in `MTree` format.

Metadata Definition

This section explains the steps required for defining and using metadata.

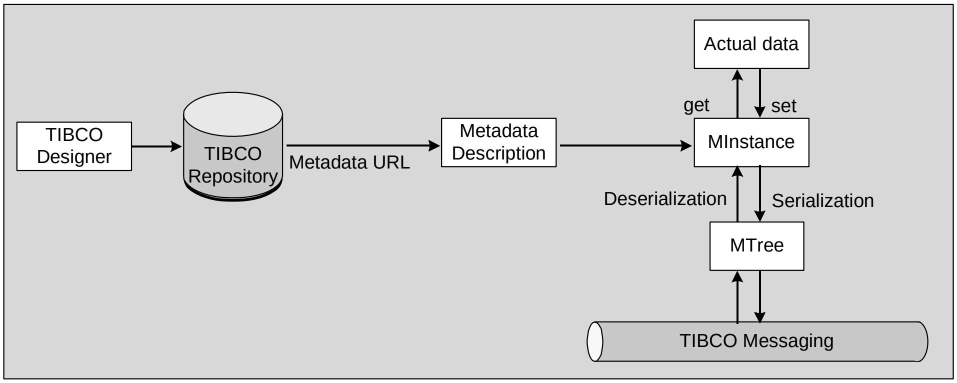
Figure 15 Metadata Creation and Usage Overview



- Defining Metadata Classes. At design-time, metadata is either defined using TIBCO Designer or by extracting information about the data the adapter works with from the target application. For data extraction, you need to write a schema tool that extracts the data from the source application and places it in the repository. Developers can also edit an AEXML file.
- Creating Classes Based on Metadata Objects. The SDK creates instances of subclasses of `MModeledClassDescription` to describe the metadata and stores it with `MClassRegistry`. By default, this happens when the custom adapter starts.
- Creating Runtime Data. At runtime, the adapter creates instances (`MInstance`) as defined by the metadata and populates the `MInstance` attributes with actual object data from the target application for publishing or responding to request-response invocations. On the receiving side, a subscriber adapter

converts the incoming message back to an instance through deserialization, which also validates the message against metadata definition.

Figure 16 Metadata Runtime Implementation



MInstance Implementation

TIBCO Adapter SDK provides the following two types of `MInstance` implementations in C++ SDK:

- `MInstanceMapImpl`: This implementation is the same as the `MInstanceImpl` in previous releases. It adopts hash tables to improve the access performance, but consumes more memory. `MInstanceMapImpl` is the default implementation.
- `MInstanceListImpl`: This implementation consumes less memory, however, the access performance degrades slightly compared with the `MInstanceMapImpl` implementation.

The `MInstanceListImpl` implementation is suitable for the following scenarios:

- The total number of attributes in `MInstance` is less than 50.
- The memory capacity is limited.
- There are large numbers of nested `MInstance` objects to be referenced by one `MInstance` object.

For a Request-Response Service, the adapter can use the method `void setInstanceImpl(MInstanceImplSelector impl)`, provided in `MRpcServer`, to choose an `MInstance` implementation.

For a Request-Response Invocation Service, the adapter can assign an instance implementation to the constructors of class `MOperationRequest`.

Defining Metadata Classes

The adapter developer defines metadata classes by using TIBCO Designer or by editing an AE XML file, which can be verified using the schema available as part of the SDK.

Using TIBCO Designer

TIBCO Designer includes a hierarchy of all metadata classes defined in the SDK. Instantiate these classes as appropriate for describing the data the adapter will work with, then save the project. For additional information, see the *TIBCO Designer Palette Reference*.

After defining the metadata classes, save them as part of the project or export them to a file.

Metadata Types

There are two types of metadata:

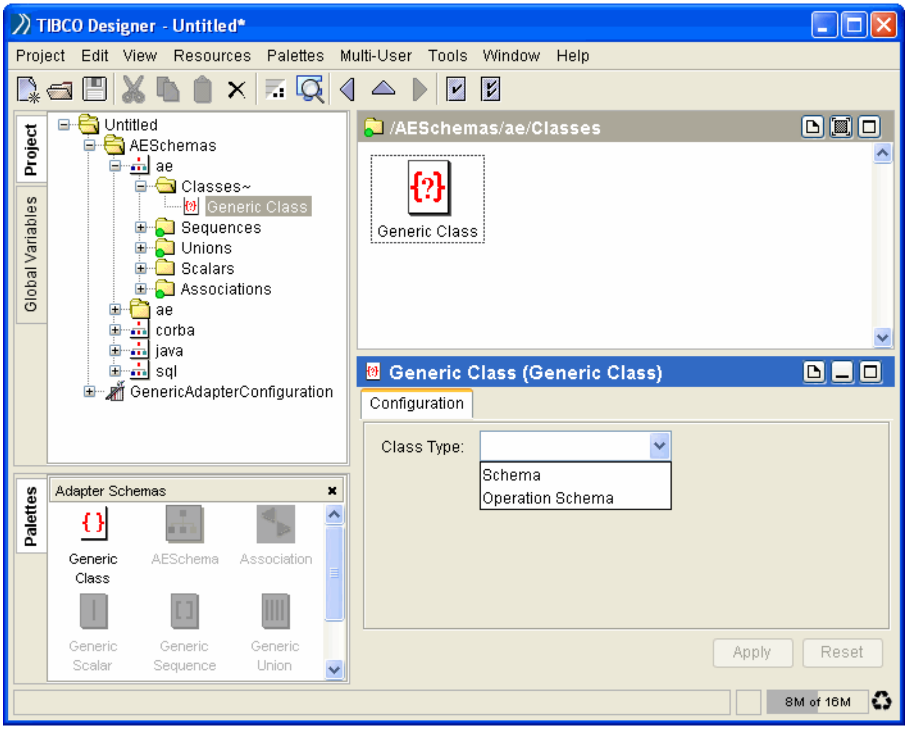
- Metadata that restricts incoming or outgoing data.

  This metadata is defined through class, union, or association resources, and has scalar type or complex type (which can be another class, union, or association) attributes.

- Metadata that restricts operation methods, parameters, and exceptions.

  This metadata should have a name, and a method or methods with a return class (if any), parameters (name, class, and the direction: in, out, or both), and any exceptions the operation raises. This metadata is used by SDK Operations classes.

Figure 17 *Defining Metadata with TIBCO Designer*



Working With the AEXML Repository File

At times, you may want to edit the AEXML repository file directly. There are several ways to create an AEXML schema file:

- Launch TIBCO Designer, select the top-level resource, then select **Resource > Export Resource**.

  In cases where adapter schema changes infrequently, you can supply preconfigured schema by manually creating it in TIBCO Designer and exporting it as an AEXML repository file. The exported file can then be imported into a new TIBCO Designer project to reuse the adapter metadata schema.

- Use the TIBCO Administrator command line export tool to specify the AEXML repository format.
- Create a tool to generate the AEXML repository from an application object or schema, then use Repository Import and Export API to load it.

The first two options export an existing repository to an AEXML file. The third option (Repository Import and Export API) creates a standalone schema configuration application for the adapter.

In cases where adapter schema varies greatly at runtime, the interactive schema configuration application will extract the metadata object from the target application and map it into an AEXML file for import into the repository. You must programmatically generate an AEXML stream by following an example export file. TIBCO Runtime Agent includes the ActiveEnterprise Schema XSD file, `AESchema.xsd`, which is used to validate an AEXML stream.

TIBCO Designer can save the adapter configuration as a template so that it can be used to create a new project, complete with schema and adapter configuration. This makes it easy to clone template for adapters that do not require a high level of schema customizations.

Creating Classes Based on Metadata Objects

The SDK parses metadata objects found in the project repository and populates the class registry. By default, this happens upon startup.

It is recommended to have all metadata data parsed upon startup. See Adapter Metadata Look-up on page 102 for instructions on how to alter this behavior.

Metadata Description Classes

Metadata description classes encapsulate the metadata description. For example:

- `MModeledClassDescription` encapsulates the descriptions of the classes themselves. It describes `MInstance`.
- `MSequenceClassDescription` encapsulates attribute types that are sequences. It describes `MSequence`.
- `MOperationParameterDescription` encapsulates operation parameters. It describes the expected inputs and outputs for an ActiveEnterprise operation.
- In the C++ SDK, `MPrimitiveClassDescription` encapsulates simple attribute type descriptions. It governs such classes as `MInteger`, `MBinary`, or `MReal`. (The Java SDK uses Java native classes in this case).

When the SDK reads the metadata from the adapter instance in the repository, each attribute must have a name and type. The SDK creates an `MAttributeDescription` for each attribute. Custom adapters can then set the attributes when they send the information over the wire. Custom adapters can also access the following information.

Table 24 Attributes Accessible to Custom Adapters

Attribute	Method
name	`MAttributeDescription::getAttributeName()`
type	`MAttributeDescription::getAttributeClassDescription()`
parent class	`MAttributeDescription::getAttributeParentClassDescription()`

Metadata Hierarchy Example

Assume you have created an `Order` class with four attributes. Three of the attributes are strings, the other, `Orderlines`, is a sequence of four attributes.

In this case, `MApp` would generate one `MModeledClassDescription` with four associated `MAttributeDescription` instances for the `Order` class, as well as one `MModeledClassDescription` with four associated `MAttributeDescription` instances for the `OrderLine` class. `MApp` also creates an `MSequenceClassDescription` consisting of one or more `OrderLine` instances.

Table 25 Metadata Hierarchy Example

Metadata	Classes
`Order` class	`MModeledClassDescription` with four associated `MAttributeDescription` classes.
`Order/ Orderlines` attribute	`MSequenceClassDescription`.
`OrderLine` class	`MModeledClassDescription` with four associated `MAttributeDescription` classes.

Creating Runtime Data

Each time information has to be sent over the wire, the custom adapter creates `MInstance` instances for those `MModeledClassDescription` instances it wants to work with. It adds attributes by iterating through the `MModeledClassDescription` attributes.

As part of instance creation, custom adapters encapsulate the attributes as instances of `MData` subclasses.

Metadata Encapsulation Classes

`MData` and its subclasses encapsulate the data itself.

- `MInstance` encapsulates class data.
- `MSequence` encapsulates sequences to be used as a type.
- `MUnion` encapsulates union data. Unions have a name and have as subelements one or more Union Member attributes. At any given time there is one, but only one, active member.
- `MInterval` encapsulates a time interval.
- `MAssocList` encapsulates association lists.
- `MChar`, `MStringData`, `MReal`, `MFixed`, `MBinary`, `MDate`, `MDateTime`, etc. support the appropriate corresponding types. See AE Schema Types and SDK Classes on page 107.

Metadata Attribute Encapsulation Classes

Metadata attribute encapsulation classes are represented differently depending on the API in use:

- In the C++ API, all metadata types, that is, all types used for the attributes of metadata elements, are represented by an SDK-defined class.
- Because Java provides encapsulated types, the Java SDK metadata interface does not include the type description classes such as `MInteger` or `MBoolean`. Instead, adapters use the native Java classes. See AE Schema Types and SDK Classes on page 107.

C++ MData Subclasses

In the C++ SDK, there are a number of `MData` subclasses for types, such as `MInteger` or `MBool`.

The C++ SDK chose this rather complex hierarchy because it facilitates introspection. Assume a custom adapter receives an `MInstance` that consists of chunks of `MData`. The design of the classes allows the custom adapter to first find out what type of data it is dealing with, for example, `MInteger`. The adapter can then downcast this `MInteger` and retrieve information about the exact type, as encapsulated by the `MPrimitiveClassDescription` data member, for example `"i4"`.

When custom adapters get ready to send data across the wire, they create instances of `MInstance` for each item defined as a class in the adapter instance. Custom adapters then set values for each instance. `MData` and its subclasses allow the custom adapter to encapsulate data at a high level. For example, you can use `MInteger` for all integer primitive types, while at the same time maintaining the information about the specific primitive type that is used.

The Java SDK does not have SDK subclasses such as `MInteger` or `MStringData` because these classes are available as part of Java.

C++ Data Encapsulation Classes

When a C++ custom adapter wants to send data or retrieve data, it uses the data encapsulation classes.

1. The classes and attributes for an adapter instance are defined using the TIBCO Designer software.
2. The SDK creates class description classes based on the metadata in the project repository.
3. The adapter accesses the class descriptions as needed.
4. The adapter creates instances and sets instance values.

   The type of the instance value is the class that encapsulates the attribute’s class in the adapter instance. For example, if the attribute had the class “i4”, the code for setting the value would use an `MInteger`.
5. The custom adapter sends the `MInstance` and the SDK serializes and publishes it.

Metadata Example

The adapter first accesses the class registry to retrieve the information about the type of data that it publishes or subscribes to.

```
//retrieve the registry
MClassRegistry* pClassRegistry = m_pMApp->getClassRegistry();
//retrieve class descripton from class registry, downcast because we’re
```

```
//working with MModeledClassDescription.
MModeledClassDescription* pMCD = MModeledClassDescription::downCast
                                 (pClassRegistry->getClassDescription());

//get the attributes for the class we just retrieved.
MMap<MString, MAttributeDescription*>* pAttributes = pMCD->getAttributes()
```

The adapter then creates the instance using one of the metadata classes. For example:

`MInstance orderInstance(this,"Order");`

The custom adapter provides the first three attribute values for the attributes of the predefined class. Note that the attributes are defined in the adapter instance description object as "`i4`". The custom adapter can set values of any integer type attribute by defining the value as an `MInteger`.

```
MInteger iOrderId = 1000;
MInteger iCustomerId = 112;

// set the order
orderInstance.set("OrderId", iOrderId);
orderInstance.set("CustomerId", iCustomerId);
orderInstance.set("ShipToAddress", MStringData("3165 Porter Dr"));
//
//The fourth attribute value is a sequence of Orderline instances. The application
//creates them as follows:
//
// create a orderline sequence
MSequence orderLineSequence(this, "sequence[OrderLine]");

//create one orderLine instance
MInstance  orderLineInstance(this, "OrderLine");
orderLineInstance.set("OrderId",  iOrderId);
orderLineInstance.set("ItemId",   MInteger(1));
orderLineInstance.set("Quantity", MInteger(345));
//append to sequence
orderLineSequence.append(&orderLineInstance );

// create another orderline
orderLineInstance.set("OrderId",  iOrderId);
orderLineInstance.set("ItemId",   MInteger(1));
orderLineInstance.set("Quantity", MInteger(345));
//append to sequence
orderLineSequence.append(&orderLineInstance );
//
//The application can then set the fourth attribute of the Order instance:
//
orderInstance.set("Orderlines", &orderLineSequence);
```

Metadata Class Names

The TIBCO Adapter SDK metadata class name used in an API call can be either the full name or the short name:

- **Full Name** Each object managed by the repository has a name, called the full name because it is fully qualified. An object's full name is the concatenation of its name with the names of all its parent directories, separated by forward slash ("`/`") characters. Names are case sensitive.

  To refer to the object `myClass`, use the full name `/tibco/public/class/ae/myClass`.

  Using full name relieves the need to specify `loadURL` in the configuration because the global path specifies exactly from where to load the class description.

- **Short Name** Each object's name is referred to as the short name of that object.

  In the above example, the short name would be `myClass`. Without the parent directories concatenated, all short names must be unique. Otherwise, SDK will use the last definition in the repository, irrespective of the `loadURL` order.

A sequence class using the `sequence[`*element*`]` name notation is a special case of short name for SDK. This notation gives a self-describing type name, so by default, even without `loadURL` specification, SDK looks for the class description under the `/tibco/public/sequence/ae` path.

Only sequences using `sequence[myclass]` need to be under this particular path. This is because the path itself tells SDK where to look for sequence element type `myclass`.

To illustrate, a sequence with following path

`/tibco/public/sequence/ae/class/ae/sequence[foo]`

means that class `foo` can be found using following path

`/tibco/public/class/ae/foo`

So this requirement is valid only if the adapter is using the SDK 2.x configuration XML notation of the name `sequence[myClass]`. Possible sources of such notation are the schema generation tool shipped with TIBCO Adapters and migrated configuration from SDK 2.x. This requirement is not valid if the schema is created through TIBCO Designer because the XML configuration for SDK specifies an `elementType` attribute, which specifies where the sequence element type is to be found. The `elementType` attribute allows the use of an arbitrary sequence class name (that is, `mySequenceClassOfFoo`).

Guidelines for Metadata Use

This section gives guidelines for metadata use.

Adapter Metadata Look-up

When configuring an adapter in TIBCO Designer, you specify metadata that restrict how the data an endpoint receives or sends should be structured.

When saving the TIBCO Designer project, the metadata are stored in the project repository. The SDK parses the metadata and creates instances of the metadata classes. These instances encapsulate the metadata information stored in TIBCO Administrator. The custom adapter can access the information through `MClassRegistry` and use it to process the available data.

Using TIBCO Designer, you can specify when the objects encapsulating the metadata should be created:

- Specify one or more `loadURL` resources so that only frequently used classes are loaded upon start-up. Multiple project repository URLs can be specified from which metadata are loaded upon startup using the `loadURL` resources inside the `Metadata URLs` folder inside the adapter's `Advanced` folder.

  Only the metadata defined in the `loadURL` resource is loaded at start-up. If no `loadURL` resources are defined, nothing is loaded at start-up. Metadata that is not specified in the `loadURL` resource will be fetched on first access and will be cached for subsequent access.

- Specify project repository URLs in which the custom adapter will search for a metadata object or attribute if its description is not yet loaded. Use the `Metadata Search URL` field in the `Startup` field of the adapter configuration object.
- Specify both a search URL and load URLs.

How the SDK Performs Metadata Look-up

When executing the `MApp start()` method, the Adapter SDK loads the metadata as follows:

- The Adapter SDK first prepends the Metadata loadURL to form a full name (defined under the *genericAdapter*`/Advanced/Metadata URLs/` folder). The Metadata loadURL specifies all the schema class definition that are to be pre-loaded on initialization and cached.

- If the Metadata loadURL is not found, the Adapter SDK loads class definition from the repository using the searchURL (this is known as a lazy fetch).

The SDK checks the URL listed under the Metadata Search URL only when the class description cannot be located in the class registry using Full Name with the prepended Metadata load URL. For example:

```
Metadata Search URL= /tibco/public/class/ae/myfolder
Metadata load URL ("Metadata URLs" folder) = /tibco/public/class/ae
Short Name = GreetingClass
```

The full name used by the SDK to locate the class description will first be

`/tibco/public/class/ae/GreetingClass`

If the class description is not found there, SDK will use

`/tibco/public/class/ae/myfolder/GreetingClass`

Restrictions on Metadata

Be aware of the following restrictions on metadata:

- The top-level class in any message sent to an adapter should be an `MInstance`.

  Inner classes may be unions, sequences, or scalars. While it is possible to create and use messages with a top level class of sequence or union, these top-level classes can be difficult to use in conjunction with mappers.

- The ActiveEnterprise schema type `any` is not a class type but an ActiveEnterprise scalar type. Type `any` should not be used in the context of the `any class` in a top level schema.

- Class names and attribute names must not use the dot (.) character.

  The dot character has special meaning in different places (such as subject name where it is used as element separator). Some adapters may decide to use a class name in their subject space, which can cause unpredictable issues.

- Schema class names and attributes should avoid using non-alphanumeric characters if at all possible.

  Schema class names cannot have parentheses '(' ')' because they are used to distinguish script functions. The '^' is used by TIBCO Adapter SDK to delimit ActiveEnterprise wire format control fields and should not be used in class names. Below are legal non-alphanumeric characters; use them only if it is not possible to implement the adapter using alpha-numerics:

  '_', ,'%','@', '|', '~', '{', '}', '#','-','$'

- Schema class names and attribute names can only be in ASCII alphanumeric characters (only schema attribute *values* support Unicode).

- Avoid the use of the `sequence[any]` data type where elements may have different classes (a heterogeneous list) because not all adapters are able to handle such list types.
- Do not define and use an `Operation` class schema that uses inheritance. The ActiveEnterprise metadata model does not implement features such as C++ class method override and virtual methods.
- Earlier versions of Adapter SDK allow to create schema that contained both simple value attributes and operation methods. Such schema are not supported in TIBCO Designer, which can read but cannot create class schema with both value attributes and operation methods. You must create separate classes for attributes and operations. This is enforced by TIBCO Designer.
- A superclass and its child classes cannot have duplicate attribute names. If a child class and its parent class have duplicate attributes, a duplicate attribute exception will be thrown.
- If a long class name (`/tibco/public/class/ae/`*className*) is specified without specifying a loadURL to access the class, and the repository file is very large and accessed remotely, an `MConstructionException` is thrown noting that the metadata class is not found.

Working with XML and XSD

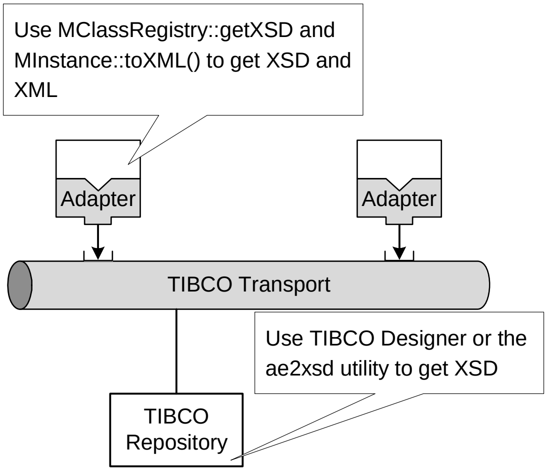
Adapter SDK supports AESchema XSD (generic XSD is not supported). AESchema XSD support is defined as the conversion of `MInstance` to and from XML. In other words, Adapter SDK can only deserialize from an AESchema based XML data into an `MInstance`. Conversely, Adapter SDK can only serialize `MInstance` into AESchema based XML data.

If you have XML data from a third party application and want that data to be part of a TIBCO application, the XML data must conform to an AESchema XSD.

Because third party XML can, probably, not be created conforming to AESchema XSD, you must transform the third party XML using either BusinessWorks or a transformation plug-in. A transformation XSLT (mapping) can be generated using an AESchema XSD generated from the schema classes defined in Designer together with the XSD of the third party XML. For more information on the transformation plug-in, refer to Transformation Plug-in on page 145. You can always bypass any Adapter SDK supported ways of handling serialization and deserialization yourself. However, that is outside the scope of this guide.

Figure 18 shows where you can access an XSD from an AESchema defined class. You may want to use this XSD in an external modeling tool.

Figure 18 Access an XSD from an AESchema Defined Class


Use MClassRegistry::getXSD and MInstance::toXML() to get XSD and XML
Adapter
Adapter
TIBCO Transport
Use TIBCO Designer or the ae2xsd utility to get XSD
TIBCO Repository

You can access the XML/XSD as follows:

1. Use the TIBCO Turbo XML tool to create the XSD. It allows you to save the XSD to a repository.
2. Export XSD from the repository using TIBCO Designer.
   - Use the **Tools > Export As XML Schemas** menu command in TIBCO Designer. See the *TIBCO Designer User's Guide* for details.
   - Use the `ae2xsd` utility to export an XSD. The utility is documented in the *TIBCO Runtime Agent Administrator's Guide*.
3. Use the following programming tools to get the XML and XSD from an application. These APIs allow you to access the XML and XSD needed for transformation or manipulation. It is useful when you want to transform non AESchema XSD based XML into SDK's AESchema based XML (AEXML).
   - Get the AESchema XSD from the Adapter SDK class registry with `MClassRegistry::getXSD(aeSchemaClassName)`.

     SDK parses the incoming AESchema XML. The XSD from `MClassRegistry::getXSD(aeSchemaClassName)` validates the AEXML data instance.
   - Get the XML string message using `MInstance::toXML()` which gives the XML serialized form of MInstance (or use `MTree::getTextBody()` if you are accessing `MTree`).

By default, `MAppProperties` is configured with `XSDGENERATION` set to `OFF`. So, by default, the XSD is not used while parsing the AEXML. If you configure `MAppProperties` with `XSDGENERATIONON` set to `ON`, Adapter SDK can parse an incoming XML message using the XSD for the given class.

The `getXSD()` method is used to access the automatically generated XSD and is automatically used in the validation process. After `XSDGENERATION` is set to `ON`, you need not explicitly call `getXSD()` to do the validation.

AE Schema Types and SDK Classes

This section discusses SDK date and time classes, then lists the mapping of type names to SDK classes for both C++ and Java.

SDK Date and Time Classes

TIBCO Adapter SDK contains a number of classes that can be used to specify date and time information.

Table 26 SDK Date and Time Classes

SDK Class	Description
`MDateTime`	By default, `MDateTime` uses the local time. When data of type `MDateTime` are sent, they are converted from the time zone used by the sender to GMT/UTC. When data of type `MDateTime` are received, they are converted from GMT/UTC to the local time zone at the location of the recipient. This may lead to confusing results, especially if the message crosses the international date line. Sent using TIBCO Rendezvous as a `TIBRVMSG_DATETIME` type for `aeRvMsg` format. `MDateTime` should be used when sender and receiver of a time value need a common reference timezone. Since the timezone is fixed at GMT/UTC, additional computation can be made on receiver side without ambiguity.
`MTime`	Contains only the time value as a string. This string time is sent "as-is" on the wire and can be of any time zone. `Time` should be used when the time value is to be kept without conversion to/from GMT/UTC.
`MDate`	Contains the time value as a string. `Date` should be used when the date is to be kept without conversion to/from GMT/UTC.

Mapping AESchema Types to C++ MData Subclasses

This section describes the mapping between class names and `MData` subclasses for the C++ SDK.

Table 27 lists the type (class) defined in the repository, the corresponding `MData` class, and whether a default value is supported.

Table 27 Mapping AESchema Types to C++ MData Subclasses

Type	MData Class	Default?	Example
any		Yes, treated as `MStringData`	"Any string here"
string	`MStringData`	Yes	"Don't Panic!"
fixed.*p.s*	`MFixed`	Yes	*p* is the total number of digits; by default unlimited *s* is the number of digits after the decimal point; by default unlimited For example, the number 123.45 would be represented by 5.2, *not* by 3.2.
boolean	`MBool`	Yes	"true", "TrUe", "false"
dateTime	`MDateTim`	Yes	"1966-04-07T18:39:09.030544", "current " means current date/time
date	`MDate`	Yes	"1994-11-05", "current" means current date
time	`MTime`	Yes	"08:15:27.4444", "current " means current time
interval	`MInterval`	Yes	"P2DT3H2M", "PT3600S"
i1	`MInteger`	Yes	"1", "127", "-128"
i2	`MInteger`	Yes	"1", "703", "-32768"
i4	`MInteger`	Yes	"1", "703", "-32768", "148343", "-1000000000"
i8	`MInteger`		"1", "703", "-32768", "148343433456534", "-1000000000000000"
ui1	`MInteger`	Yes	"1", "255"
ui2	`MInteger`	Yes	"1", "255", "65535"
ui4	`MInteger`	Yes	"1", "703", "3000000000"
ui8	`MInteger`	Yes	"1483433434334"
r4	`MReal`	Yes	".31415E+1"
r8	`MReal`	Yes	".3141159265358979E+1"

Table 27 Mapping AESchema Types to C++ MData Subclasses (Cont'd)

Type	MData Class	Default?	Example
binary.*n*		not supported	
char.*n*	`MChar`	Yes. By default *n* = 1.	"ABCD"
sequence	`MSequence`	not supported	
<custom class name>	`MInstance`	not supported	
<custom union name>	MUnion	not supported	

If using Latin-1 encoding, char.4 would be 4 bytes. If using UTF-8, char.4 would be 8 bytes.

If you specify char.4 using Latin-1 encoding, there will be padding when the full length of the type is not used. For UTF-8, there will be no padding.

Prespecifying Data Length

For some types, such as `char` or `binary`, you can pre specify the length of the data in the repository, then refer to that prespecified bounded data type. For example, `binary.10` refers to binary data containing at most 10 bytes.

Mapping AESchema Types to Java Classes

Table 28 lists how the types you specify as an attribute's class in the repository map to classes in an SDK application. Most of the types map directly to a corresponding Java class. A few others map to a class defined by the SDK.

Table 28 Mapping AESchema Types to Java Classes

Type	Java	Example
any	java.lang.Object	"Any string here"
string	java.lang.String	"Don't Panic!"

Table 28 Mapping AESchema Types to Java Classes (Cont'd)

Type	Java	Example
fixed.p.s	java.math.BigDecimal	*p* is the total number of digits; by default unlimited *s* is the number of digits after the decimal point; by default unlimited For example, the number 123.45 would be represented by 5.2, *not* by 3.2.
boolean	java.lang.Boolean	"true", "false"
dateTime	java.util.Date	"1966-04-07T18:39:09.030544"
date	java.util.Date	"1994-11-05"
time	java.util.Date	"08:15:27.4444"
interval	com.tibco.sdk.metadata.MInterval	"P2DT3H2M", "PT3600S"
i1	java.lang.Byte	"1", "127", "-128"
i2	java.lang.Short	"1", "703", "-32768"
i4	java.lang.Integer	"1", "703", "-32768", "148343", "-1000000000"
i8	java.lang.Long	"1", "703", "-32768", "148343433456534", "-1000000000000000"
ui1	java.lang.Byte	"1", "255"
ui2	java.lang.Short	"1", "255", "65535"
ui4	java.lang.Integer	"1", "703", "3000000000"
ui8	java.lang.Long	"1483433434334"
r4	java.lang.Float	".31415E+1"
r8	java.lang.Double	".31411592653589 79E+1"
binary.n	byte[]	
char.n	java.lang.String	"ABCD"
char.1	java.lang.String (with length=1) or java.lang.Character	"A"
sequence	com.tibco.sdk.metadata.MSequence	

Table 28 Mapping AESchema Types to Java Classes (Cont'd)

Type	Java	Example
<custom class name>	com.tibco.sdk.metadata.MInstance	
<custom union name>	com.tibco.sdk.metadata.MUnion	

Table 29 provides more detail on the types that are currently supported for attributes.

Table 29 Types Supported for Attributes

Type	Description	Example
m_any	Unspecified type - can model Java Object, C++ void *, COM VARIANT.	
m_string	String of unlimited length.	Don't Panic!
m_fixed.*p.s*	Fixed number. Precision (*p*) is the total number of digits. Scale (*s*) is the number of digits to the right of the decimal point and must be less than or equal to the precision.	12.0042
m_boolean	"0" or "1"	0, 1 (1 == "true")
m_dateTime	A date in a subset of ISO 8601 format, with optional time and no optional zone.	2088-04-07T18:39:09
m_date	A date in a subset ISO 8601 format. (no time)	2094-11-05
m_time	A time in a subset ISO 8601 format, with no date and no time zone. Fractional seconds may be as precise as nanoseconds.	08:15:27.4444
m_interval	Time interval	P2DT3H2M, PT3600S
m_i1	1 byte signed int	1, 127, -128
m_i2	2 byte signed int	1, 703, -32768
m_i4	4 byte signed int	1, 703, -32768, 148343, -1000000000
m_i8	8 byte signed int	1, 703, -32768, 14834343456534, -1000000000000000
m_ui1	1 byte unsigned int	1, 255

Table 29 Types Supported for Attributes (Cont'd)

Type	Description	Example
m_ui2	2 byte unsigned int	1, 255, 65535
m_ui4	4 byte unsigned int	1, 703, 3000000000
m_ui8	8 byte unsigned int	1483433434334
m_r4	Real number with 7 digits of precision.	.31415E+1
m_r8	Real number with 15 digits of precision.	.3141159265358979E+1
m_binary.*n*	Binary. length (*n*) may be specified. Default is unlimited.	binary binary.32
m_char.*n*	Character string, *n* characters long. Length may be specified. Default is 1.	char (==char.1) char.64

Chapter 7 TIBCO ActiveEnterprise Operation Model

This chapter discusses how the TIBCO Adapter SDK implements the ActiveEnterprise operation model.

Topics

Overview

The ActiveEnterprise operation model allows custom adapters to perform operation invocation or to act as an operation server. The model provides a simple way to represent and use request/reply interactions between distributed invocation components.

This document uses the term ActiveEnterprise operation model instead of Remote Procedure Call (RPC) because invocation of ActiveEnterprise operations can be remote or in-process.

The ActiveEnterprise operation model enhances basic TIBCO Rendezvous and TIBCO Enterprise Message Service request/reply with the following features:

- Operations are specified using TIBCO Designer and saved in the project repository. The operation specification can be shared across ActiveEnterprise products and introspected at runtime for dynamic invocation.
- The SDK enforces proper marshalling/unmarshalling of the requests and replies and therefore limits errors in corresponding code.
- The ActiveEnterprise operation model provides a framework for passing parameters in both directions and for allowing servers to throw exceptions.
- The ActiveEnterprise operation model optimizes invocations when an in-process server implementation is available.

The SDK includes a number of examples for remote operations in the *SDK_HOME*/`examples` directory.

ActiveEnterprise Operations

This section explains the synchronous and asynchronous ActiveEnterprise operations, as well as how to implement them.

Synchronous and Asynchronous

Both client and server can be either synchronous or asynchronous.

The terms *synchronous* and *asynchronous* have special meaning in this context, as defined in this section.

- A *synchronous client* invokes an operation and then waits for the specified time for the reply to arrive.
- An *asynchronous client* invokes an operation and specifies a reply listener class, which will be used to process the reply. Using asynchronous invocation, an adapter can invoke multiple remote operations without having to wait for the reply.
- A *synchronous server* provides an implementation where the reply can immediately be built and sent back.
- An *asynchronous server* provides an implementation where the reply is built asynchronously. When the reply is ready, it is sent back to the client.

The synchronous or asynchronous characteristics are inherently private to the client or server implementation. Synchronous or asynchronous clients are fully inter-operable with either asynchronous or synchronous servers.

Supported Invocation Protocols

The SDK allows adapters to use remote and in-process operation invocation.

- Remote Invocation—Client and server communicate across a network.
- In-Process Invocation—Whenever the server and the client reside in the same `MApp`, the invocation is optimized to prevent unnecessary communication overhead. In that case, the SDK does not broadcast requests to anyone outside that process and does not send out messages.

Invocations With or Without Replies

Most ActiveEnterprise operations, regardless of whether they use remote or in-process invocations, expect a reply. The reply can be expected synchronously or asynchronously (see Synchronous and Asynchronous on page 115).

The SDK also supports one-way invocation. When an operation is declared as one-way, no reply is expected by the client.

Implementing ActiveEnterprise Operations

Implementing ActiveEnterprise operations consists of the following tasks:

- Use the TIBCO Designer software to:
  - Configure the operations to be performed using TIBCO Designer `Schema` resources. See *TIBCO Designer Palette Reference*.
  - Specify the services using the `TIBCO Designer Generic Adapter Configuration`. If necessary, customize the endpoints and sessions that are automatically created. See *TIBCO Designer Palette Reference*.
- Use SDK classes to implement the behavior in the custom adapter code.
  - The client builds a request instance based on the operation schema information and invokes the corresponding operation. The client receives a reply in the form of a reply instance.
  - The server registers an implementation for each operation based on the schema information. The server implementation receives a request instance object and builds a reply instance object.

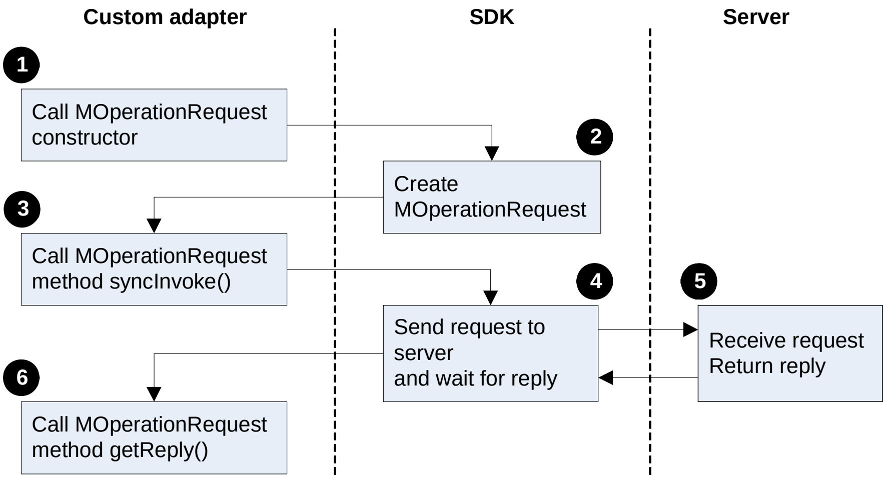
Implementing ActiveEnterprise Operations in C++

This section demonstrates the control flows for a synchronous or asynchronous client and a synchronous or asynchronous server. Code examples can be found in the `operation` sample program.

Synchronous Client Control Flow

Synchronous clients use the control flow shown below.

Figure 19 Control Flow for Synchronous Clients


Custom adapter
SDK
Server
1
Call MOperationRequest constructor
2
Create MOperationRequest
3
Call MOperationRequest method syncInvoke()
4
5
Send request to server and wait for reply
Receive request Return reply
6
Call MOperationRequest method getReply()

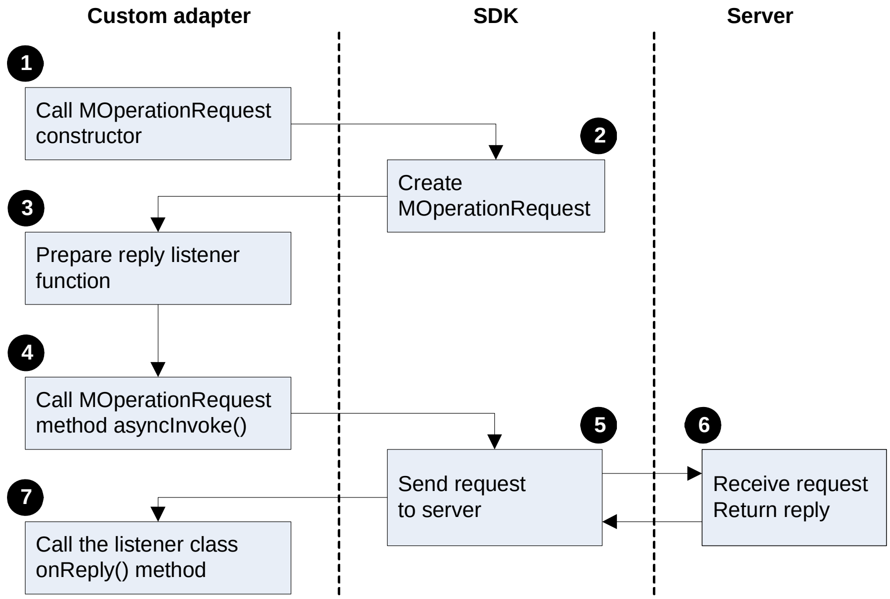
1. The custom adapter calls the `MOperationRequest` constructor.
2. The SDK creates the `MOperationRequest` instance.
3. The custom adapter calls `MOperationRequest::syncInvoke()`.
4. In response, the SDK sends the request from the client to the server.
5. The server returns an `MOperationReply`. The server can be either synchronous or asynchronous.
   - — A *synchronous server* provides an implementation where the reply can immediately be built and sent back
   - — An *asynchronous server* provides an implementation where the reply is built asynchronously. When the reply is ready, it is sent back to the client.

6. The custom adapter calls the `MOperationRequest::getReply()` method to retrieve the arguments and continue work.

Asynchronous Client Control Flow

Asynchronous clients use the control flow shown below. The step 3, step 4, and step 7 are different from the synchronous client.

Figure 20 Control Flow for Asynchronous Clients


Custom adapter
SDK
Server
1
Call MOperationRequest constructor
2
Create MOperationRequest
3
Prepare reply listener function
4
Call MOperationRequest method asyncInvoke()
5
Send request to server
6
Receive request Return reply
7
Call the listener class onReply() method

1. The invoking application calls the `MOperationRequest` constructor.
2. The SDK creates the MOperationRequest instance.
3. The custom adapter prepares the reply listener function.
4. The custom adapter calls the `MOperationRequest::asyncInvoke()` method.
5. In response, the SDK sends the request from the client to the server.
6. The server returns an `MOperationReply()`.
7. The custom adapter calls `onReply()` in the reply listener class.

Synchronous Server Control Flow

To implement a synchronous server:

1. In the TIBCO Designer software, define a Request-Response Service and specify the transport type (TIBCO Rendezvous or TIBCO Enterprise Message Service). The corresponding endpoints and sessions are generated automatically.
2. Define subclasses of `MOperationImpl` and define the `onInvoke()` method. This method is called in response to a client request.
3. Create instances of adapter-defined subclasses of `MOperationImpl` using the class and operation name and the server name declared in the Server endpoint object.
4. Bind each `MOperationImpl` instance to the appropriate server using the *servername* parameter of the constructor.

Asynchronous Server Control Flow

Implementing an asynchronous server is similar to implementing a synchronous server. In C++, the custom adapter must, however, take ownership of the `Reply` handle so that the object is not deleted.

Implementing ActiveEnterprise Operations in Java

The remote operation paradigm in the Java SDK is similar to that in the C++ SDK, with two exceptions:

- There are `MClientRequest/MClientReply` and `MServerRequest/MServerReply` classes (instead of just `MOperationRequest` and `MOperationReply`)
- In addition to using any of the `onInvoke()` methods, Java programmers can take advantage of the Java introspection mechanism to use an adapter-defined method with a name of their choice.

This section gives an overview of ActiveEnterprise operation programming, which is also illustrated by the `zapadapterRPC` example program.

Defining ActiveEnterprise Operation Elements

Using TIBCO Designer, you define the clients, servers, and operations in two different locations:

- Remote invocation operations, their parameters, and the exceptions they raise are defined in the `Schemas` folder.

  See the *TIBCO Designer Palette Reference* available via **Help > Designer Help** from TIBCO Designer.

- Protocol information for both client and server is defined as part of the `Generic Adapter Configuration`.

  See the *TIBCO Designer Palette Reference* available via **Help > Designer Help** from TIBCO Designer. See also Defining Endpoints and Protocols on page 121.

The following example shows the information required to define an ActiveEnterprise operation.

```
Class Name = rfc-order
    Operation Name = rfc-get-order
    Returns= string
        Parameter Name = inp1
        Parameter Type = i4
        Direction = in
        Default = 4

        Parameter Name = inoutp2
        Parameter Type = i4
        Direction = inout
        Default = 5

        Parameter Name = outp3
Parameter Type = i4
Direction = out

Raises Name = order-exception
Type = string
```

Defining Endpoints and Protocols

The endpoint and protocol information is defined using TIBCO Designer. For example, the following information is required to define an endpoint named `client1` and an endpoint named `server1`.

```
Endpoint Name = client1
         Type = RV RPC Client
      Session = session1
      Subject = OPERATION.HELLO
Invocation Timeout = 5000

     Endpoint Name = server1
              Type = RV RPC Server
           Session = session1

           Subject = OPERATION.HELLO
```

Note the following points while defining endpoints:

- There is no `RvCmqClient` in this example because certified remote operations are implemented with an `RvCmClient`.
- Client and server must use the same subject.

Defining and Invoking the Methods

In the Java SDK, there are two ways to implement the method that is executed remotely.

- Using the onInvoke() or onOnewayInvoke() Method, page 121
- Using a Method of Your Choice to be Invoked Remotely, page 122

Using the onInvoke() or onOnewayInvoke() Method

To use one of the SDK-supplied methods:

1. Create a subclass of `MOperationImpl`.
2. Create a method that you want to be invocable by a client through the remote operation service. Override either `onInvoke()` or `onOnewayInvoke()` to provide a meaningful implementation (depending on whether the operation is defined as one-way).

3. If you choose to implement `onInvoke()`, an `MServerReply` object is passed to the callback method. The custom adapter is responsible for calling `reply()` on this object to send back the reply. The resulting behavior is that of an asynchronous server.

Using a Method of Your Choice to be Invoked Remotely

To allow for a more natural syntax, you can also use a method of your choice.

1. Create an `MOperationImpl` subclass.
2. Define a Java method whose name and parameter signature match the operation description provided in the repository.
   - The parameter list must consist of in and in-out parameters in the order defined.
   - By default, the returned object is assumed to be the returned value of the operation.
   - You can also choose to have the returned value of the Java method be of type `MServerReply`. In that case, the `MServerReply` object can hold the returned value and/or the in and in-out parameter values. If you have in-out parameters in the operation description, return `MServerReply` from the operation method.
   - Exception of any type can be thrown whenever appropriate; however, only exceptions of type `MOperationException` are caught and forwarded back to the requestor. Other exceptions are just logged locally.

The resulting behavior is that of a synchronous server.

> Whether a reply is sent back depends on the operation's one-way property. Nevertheless, the custom adapter must not call `reply()` on the `MServerReply` object even if it uses it to pass back information to the client. The SDK sends back the reply if appropriate.

If a method as described above exists (determined by class introspection), it takes precedence over either `onInvoke()` or `onOnewayInvoke()`.

Chapter 8 Advanced Features

This chapter discusses advanced features of the TIBCO Adapter SDK.

Topics

Multithreaded Adapters

The TIBCO Adapter SDK allows flexible, platform-independent multithreading. This section discusses the following aspects of multithreaded adapters:

- Deciding on Multithreaded Implementation, page 124
- Multithreading and MDispatcher, page 125
- Multithreading Scenarios, page 126
- Writing a Multithreaded Adapter with the C++ SDK, page 127
- Writing a Multithreaded Adapter with the Java SDK, page 128

Deciding on Multithreaded Implementation

A thoughtful multithreaded design may help a custom adapter achieve a greater level of concurrency and performance in terms of latency.

⚠ Consider carefully whether the custom adapter requires multiple threads. Threads result in a more complex programming logic and are therefore more error-prone and more difficult to debug. In addition, a multithreaded SDK-based adapter consumes more resources than a single-threaded one on a single-CPU machine.

- Use multithreading in the following situations:
  - For custom adapter that uses blocking synchronous methods, is deployed on multi-CPU platform, or both.

    To fully exploit a multiprocessor system, the custom adapter running on it should also be multithreaded.

    Multithreading is useful for a server that takes one request at a time, and then spends most of its time blocking on I/O. In this case, the server would have unacceptable performance running as a single-threaded application. With a pool of threads, each thread can work on one request and several requests can be handled simultaneously, making use of otherwise idle processor cycles.
- Do not use multithreading in the following situations:
  - If it is not needed.

    Processing resources may not be the only factor hindering adapter performance. The interface supported by the target application can also be a bottleneck that eliminates any benefit resulting from multithreading the adapter. Multithreading cannot address performance issues caused by system resource bottlenecks.

- If messages must be processed in the order sent.

  While TIBCO Messaging delivers messages in the order sent to the adapter on a single transport, a multithreaded adapter may not process the messages in the order received.

- If the target application cannot accept multiple threads.

All C++ custom adapters that implement multithreading must set the multithread behavior in C++ SDK by calling the `MAppProperties::setMultiThreaded()`.

Multithreading and MDispatcher

It is recommended to make an adapter multithreaded using the TIBCO Adapter SDK `MDispatcher` class. The different constructors for `MDispatcher` allow you to associate each instance with either an `MApp` application manager or with an `MApp` and an `MSession`.

Custom adapters can create an instance of `MDispatcher` to spawn a thread that dispatches events for a given session. The `MDispatcher` class can be used with both `MRvSession` and `MJmsSession`.

Creating a multithreaded adapter is typically a matter of determining which session requires additional threads and constructing the appropriate `MDispatcher` objects.

`MDispatcher` is used in conjunction with `MSubscriber` but not with `MPublisher`.

Other SDK Objects

In addition to the new `MDispatcher` class, the following classes can be used in a multithreaded fashion:

- `MClassRegistry` is thread safe and thread aware. As a result, you can serialize and deserialize across threads.
- `MTrace` is thread safe and thread aware. You can therefore perform tracing across threads.
- The SDK may need to handle events that are not attached to any explicit `MSession`, for example, TIBCO Hawk messages or internal events. These events must be handled through the main `MApp` event manger with the `MApp::nextEvent()` call.

If `MApp::start()` is called with no arguments or with `Mtrue`, the SDK handles these events automatically. The correct synchronized shutdown behavior of a multithreaded adapter depends on dispatching these events. If an adapter fails to ensure that `MApp::nextEvent()` is called during shutdown, the behavior of the SDK is undefined.

Multithreading Scenarios

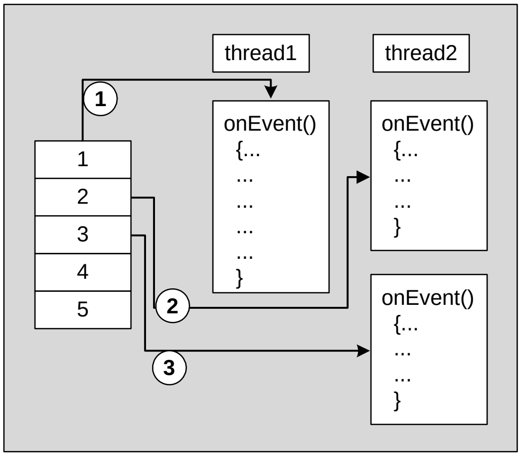
This section gives an overview of multithreading scenarios.

Single Event Queue

If all the dispatchers you created are associated with one session (and therefore with a single event queue), the multithreading adapter program proceeds as follows:

1. Thread 1 picks the first item of the queue and executes the `onEvent()` method.
2. When the next thread, thread 2, becomes available, it picks the second item and executes the `onEvent()` method.
3. When the next thread, again thread 2, becomes available, it picks the third item and executes the `onEvent()` method.

Figure 21 Multithreading for Single Event Queue



Queue Group

If you are using a queue group, there are two options for handling multithreading.

- Associate the `MDispatcher` with `MApp`. This gives each session equal weight when a thread becomes available.
- Associate `MDispatcher` instances with individual sessions managed by `MApp`.

You can also use a combination approach, that is, associate some `MDispatchers` with `MApp`, then create additional dispatchers and associated it with a session that has priority over all other sessions.

Writing a Multithreaded Adapter with the C++ SDK

To write a multithreaded C++ custom adapter:

1. When setting up the `MAppProperties` instance for the adapter, call `MAppProperties::setMultiThreaded();` to turn on multi-threading.

   You can turn multi-threading off by calling `MAppProperties::setMultiThreaded (MFalse)`.

   The C++ SDK is single-threaded by default and will only behave correctly for multithreaded custom adapters that make this call. The following example fragment from `mt_rpc` shows how the call is made as part of `main()`.

```
int main(int argc, char** argv) {
   MAppProperties appProperties;
   appProperties.set( MAppProperties::APPNAME, "mt_rpc");
   appProperties.set( MAppProperties::APPVERSION, "5.0.0");
   appProperties.set( MAppProperties::APPINFO, "SDK multi-threaded RPC example");
   appProperties.set( MAppProperties::APPCONFIGURL,
                       "/tibco/private/adapter/examples/mt_rpc/mt_rpc_client");
   appProperties.set( MAppProperties::APPREPOURL, "tibcr://CPP_EXAMPLES");
   appProperties.setMultiThreaded();
   OrderAdapter myAdapter(appProperties);
   myAdapter.start();
}
```

2. In `MApp::onInitialization()`, construct as many instances of `MDispatcher` as required.

   In the example below, the dispatchers are associated with a session.

```
void OrderServer::onInitialization() throw(MException){
   // create dispatchers which will handle event dispatching in a
   // platform-independent way for a given MRvSession
   m_pDispatchers = new MList<MDispatcher *>();
```

```
MRvSession *pSession = MRvSession::downCast(
                           this->getComponentByName(RVSESSION_NAME) );
m_NumDispatchers = 5;
if ( pSession ) {
   for (unsigned int i=0; i<m_NumDispatchers; ++i ) {
      MDispatcher *pDispatcher = new MDispatcher(this, pSession );
      m_pDispatchers->push_back( pDispatcher );
   }
}
printf( "RPC Server - Ready with %u MDispatchers\n",
                      m_NumDispatchers );
```

}

The custom adapter must also handle inter-thread communication and any locking concerns explicitly.

Writing a Multithreaded Adapter with the Java SDK

The `mt_pubsub` demonstrates how to write a multithreaded adapter using `MDispatcher`. Each time an instance of `MDispatcher` is created, a new thread is spawned to dispatch incoming events on the relevant `MSession`. The additional threads allow the subscriber to remain responsive facing lengthy operations.

1. In the `onInitialization()` method, create instances of `MDispatcher`, as follows. The dispatchers are associated with a session.

```
protected void onInitialization() throws MException
   {
      MComponentRegistry registry = getComponentRegistry();

MSubscriber sub = registry.getSubscriber("sub");
      if (null == sub) {
         throw new MException("PUBSUB-0003", "sub");
      }
      // attach an event listener to the subscriber
      sub.addListener(new DataEventHandler());

      // create 5 dispatchers for the session that is associated
      // with the subscriber
      MSession session = sub.getSession();
      for (int i = 0; i < NO_OF_DISPATCHERS; ++i) {
         dispatchers.add( new MDispatcher(this, session) );}
   }
```

2. In `onTermination()`, you must stop all the dispatchers created earlier.

```
protected void onTermination() throws MException
   {
```

```
// stop all the dispatchers that were created during
//onInitialization

System.out.println(new Date() + " Stopping all the
dispatchers...");
MDispatcher dispatcher = null;
for (int i = 0; i < NO_OF_DISPATCHERS; ++i) {
   dispatcher = (MDispatcher)dispatchers.get(i);
   dispatcher.stop();
}
System.out.println(new Date() + " Stopped all the
dispatchers");
```

Adapter SDK Unicode Support

The C++ API and the Java API both support Unicode for application data. This allows custom adapters to work with Unicode strings programmatically and to send and receive data between applications that use a variety of supported encodings. The data can be aggregated and serialized and then be sent over the network.

The Java Adapter SDK takes advantage of native Java Unicode support.

The C++ Adapter SDK includes the `MChar` and `MStringData` classes to encapsulate Unicode data. Their constructors allow specifying the encoding for the source data. A complete list of supported encodings can be found in `MEncoding.h`. You can also create an `MWString` instance from either class, which allows you to call string manipulation methods against your data.

The following classes are available in the C++ Adapter SDK:

- `MString` and `MWString` are used when you need string manipulation methods to operate on the data.

  `MString` can encapsulate single-byte character data, while `MWString` encapsulates Unicode (UTF-16) characters.

  You must convert `MString` and `MWString` to `MChar` and `MStringData` before sending them on the network.

- `MStringData` and `MChar` are used to encapsulate Unicode data.

  Any source data strings are converted to Unicode by the Adapter SDK upon construction, as long as the source is in an encoding supported by the Adapter SDK and the encoding is provided to the constructor.

Prespecifying Encoding

Custom adapters based on the TIBCO Adapter SDK automatically configure themselves to send or receive messages in ASCII/Latin-1 or in UTF-8 wire format, depending on how the associated server-based repository instance is configured.

If you are using TIBCO Designer 5.1.2 or later to prepare the configuration, you can set the encoding directly as an attribute of the adapter configuration.

The conversion from the internal Unicode data (in UTF-16) to the wire format encoding is accomplished by the serialization to an `MTree` instance.

Two adapters based on the SDK can communicate only if they use the same encoding on the wire. A problem arises if one adapter sends Latin-1 encoded messages to another adapter expecting UTF-8 encoded messages. Since the second adapter is expecting UTF-8 on the wire, Latin-1 characters are interpreted incorrectly.

SDK C++ API takes the encoding value from the repository and no other regional settings can affect the value of encoding. There can be only one encoding per process or application. If multiple `MApp` application managers are running inside an SDK adapter, and each `MApp` connects to a different repository, all the repositories must have the same encoding value.

SDK-Internal C++ Unicode Type Conversion

This section gives an overview of how the C++ SDK performs conversion.

Internally, the C++ SDK first decides to use one of two native implementations: Latin-1 for single-byte characters or UTF-16 for double-byte characters. Whether the SDK attempts conversion, and what conversion the SDK attempts depends on the encoding argument presented to the constructor for `MChar` or `MStringData`.

- If the encoding presented to the constructor is ASCII, Latin-1, or UTF-16, no conversion is needed.

  For all other cases, the SDK attempts a best-case conversion. If conversion is required (for example, UTF-16 to Latin-1), a replacement character is used for unmappable characters.

- If Unicode conversion to and from arbitrary encodings is required, a file (`tibicudata32.dat)` containing a lookup table is required.

  Set the environment variable `TIB_ICU_DATA` to point to the directory that contains the `tibicudata32.dat` file. You need to set the variable manually. If SDK cannot find this file, it will throw an exception when you attempt to convert certain types of string encodings.

You can find the `tibicudata32.dat` file in the TIBCO Runtime Agent `config/g11n` directory. This directory also contains a `tibicudata.dat` file for backward compatibility with versions prior to SDK 5.3.

This release of SDK uses the ICU (International Components for Unicode) 3.2 version. Some common aliases from ICU are shown in the ICU Converter Explorer available at http://icu-project.org/icu-bin/convexp.

However, to maintain backward compatibility, only Adapter SDK encoding types listed in the *SDK_HOME*`\include\MEncoding.h` should be used, *not* common alias names listed on the site above.

Specifying the Wire Format Encoding

The wire format encoding for messages affects all communications for adapter applications. Either Latin-1 or UTF-8 is supported as the wire format encoding when the adapter application is using a server-based project repositories.

If the project uses only ASCII or Latin-1 data, you can set the encoding to be Latin-1, which makes the custom adapter run faster. Otherwise, use UTF-8.

Specifying Encoding for Server-Based Repositories

All project repositories managed by a particular administration server use the same encoding. You can specify the wire format encoding as a server property (`repo.encoding`) in the `tibcoadmin.tra` file. To change the wire format encoding, shut down the server, edit the `tibcoadmin.tra` file and then restart the server.

One reason for choosing a particular encoding may be consistency with another TIBCO application that uses a fixed encoding.

Specifying Encoding for File-Based Repository

At some stages of a project, you may use a file-based repository. In that case, the encoding can be set in the project repository file itself. You can make the change using the Repository Finder in TIBCO Designer or editing the `.dat` file directly.

Add the instance property below:

```
<instanceInfoProperty name="encoding" value="desired_encoding"/>
```

Note that if this repository instance is later managed by a repository server, the encoding used by the server overrides the encoding of the file.

The encoding only affects communication, it has no effect on the persistent storage of the data. TIBCO Administrator stores data in UTF-8 format regardless of the wire format encoding being used.

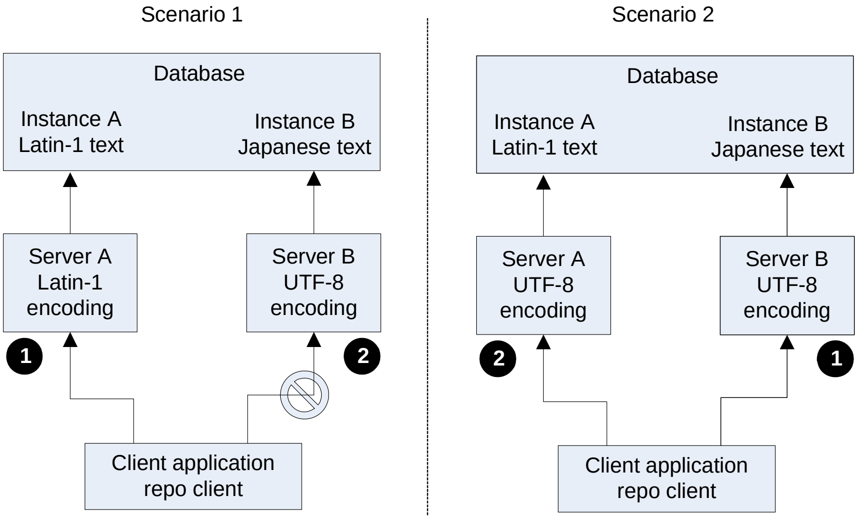
How TIBCO Administrator Determines Encoding

When an adapter application starts, the TIBCO Administrator client library forces both the instance name and discovery subject to conform to ASCII so that communication works with either encoding.

When the client is actually connecting to a server-based project repository for the first time, the encoding used by the server for that instance determines the encoding type for all TIBCO messages. The server encoding is determined by the `repo.encoding` parameter in the `tibcoadmin.tra` file.

All communicating applications must use the same wire format encoding. Therefore, all project repositories in use by applications that communicate with each other must use the same encoding. To understand the use of encoding formats, consider the following scenarios.

Figure 22 Scenarios of Encoding Formats



A client application with an embedded TIBCO Administrator client attempts to connect to two administration servers. In Scenario 1, the two servers use different encodings. In Scenario 2, the two servers use the same encodings.

The components interact as follows:

1. The client application discovers available administration servers and instances by sending a discovery message on the network. In the message, the server name, instance name, and discovery subject are restricted to ASCII characters only.
2. The client discovers servers and two repository instances.
3. In Scenario 1, the client application connects first to InstanceA through Server A. Server A uses Latin-1 as the `repo.encoding` property because the text is Latin-1. The client is now forced to use Latin-1 as the wire format encoding.

   When the client attempts to connect to Server B (which is using UTF-8), an exception is signalled because a lossy conversion would result.
4. In Scenario 2, the client application connects first to Server A (which is using UTF-8). When the client then attempts to connect to Server B, it succeeds.

Once the client's encoding is established, an exception is thrown when trying to connect to a server that uses a different encoding.

Preregistering a Subscription Service

TIBCO Rendezvous supports preregistration for RVCM sessions. In some situations, a sending RVCM session can anticipate the request for certified delivery from a (listener) persistent correspondent that has not yet registered.

Consider an example in which a database program (DB) records all messages with the subject `STORE.THIS`. The program DB creates an RVCM session that instantiates a persistent correspondent named DB_PER. All programs that send messages with the subject `STORE.THIS` depend on this storage mechanism.

One such sending program is JAN. Whenever JAN starts, it can anticipate that DB_PER will request certified delivery of the subject `STORE.THIS`. Suppose that JAN starts, but DB is not running, or a network disconnect has isolated JAN from DB. Anticipating that it will eventually receive a registration request for `STORE.THIS` from DB_PER, JAN makes an add listener call.

The sending RVCM session in JAN behaves as if it has a certified delivery agreement with DB_PER for the subject `STORE.THIS`; it stores outbound messages (on that subject) in its ledger. When DB restarts, or the network reconnects, the sender RVCM session in JAN automatically retransmits all the stored messages to DB_PER.

It is not sufficient for a sender to anticipate listeners; the anticipated listening programs must also require old messages when they create RVCM sessions.

Two methods in the SDK support this behavior:

- `MPublisher::preRegisterListener()` and `MPublisher.preRegisterListener()`.
- `MPublisher::unRegisterListener()` in and `MPublisher.unRegisterListener()`.

See the API documentation for more information.

The SDK supports endpoint types (TIBCO Rendezvous publishers and subscribers, TIBCO Rendezvous CM publishers and subscribers) that result in preregistration of the specified listener.

- If you changed the subject through the `MPublisher` method `setDestination()`, the publisher with the new subject will be unaware of any preregistration done previously through the API or the TIBCO Designer software.
- For TIBCO Enterprise Message Service sessions, it is possible to unregister an inactive durable subscriber by calling `MPublisher.unRegisterListener()`.

Setting Data to NULL Explicitly

Custom adapters can set attribute values in `MInstance`, `MAssocList` and `MOperationRequest`, and in `MOperationReply` (C++) or `MOperation` (Java) explicitly to `NULL` using a `setNullData()` method. Explicitly setting an attribute to `NULL` is different from not setting the attribute at all.

Before retrieving an attribute, a custom adapter can verify whether the attribute was set to `NULL` explicitly by calling `isNullData()`.

The following table illustrates what `get()` and `isNullData()` return depending on how data was set.

Table 30 get() and isNullData()

	Result of calling get()	Result of calling isNullData()
set value calling `set()`	`MData` value	`MFalse`
Do not set value (or unset)	Default value if there is one, `NULL` otherwise	`MFalse`
set value calling `setNullData()`	`NULL`	`MTrue`

To understand data in the SDK context, it is useful to consider both the format in which data is encapsulated and the format in which it is sent and received.

There are two primary data formats:

- `An MInstance` allows hierarchical data representation and depends on a predefined class description (metadata information). This metadata information is provided in the adapter configuration and then encapsulated in a number of classes by the `MApp` application manager.
- An `MTree` is a data structure that can be sent over a network. An `MTree` represents data as a hierarchical tree in which each node contains one or more name-value pairs.

In SDK versions earlier than 5.x, custom adapters explicitly serialized an `MInstance` to get an `MTree`. In SDK 5.x, a publisher serializes an `MInstance` when you send it. A subscriber has to deserialize the `MTree`.

TIBCO Adapter Wire Formats

This section discusses about the wire format used by SDK adapters.

⚠ Adapters should not attempt to simulate the wire format by adding control information. Experience has shown that the results are not satisfactory.

Wire Formats and Message Formats

The term *wire format* refers to the message formatting convention used by the adapter. Wire formats include `rvMsg`, `aeRvMsg`, and `aeXml`.

The term *message format* refers to a combination of the transport (TIBCO Enterprise Message Service or TIBCO Rendezvous) and the wire format. In ActiveEnterprise 5, the message format (not the wire format) is the critical attribute of an endpoint. See Message Formats on page 57. SDK-based adapters can only exchange data if the sending and the receiving application use the same message format.

- `rvMsg` wire format is used for the basic TIBCO Rendezvous message format. The SDK does not perform validation, but if you use this format, SDK adapters are compatible with non-SDK adapters. This format can be used in conjunction with TIBCO Rendezvous.

  For `rvMsg` wire format, a message of type `TIBRVMSG_MSG` is expected. If a sending application (for example, `rvsend`) publishes a simple `TIBRVMSG_STRING`, the SDK passes an `MExceptionEvent` to the event listener.

- `aeRvMsg` is the TIBCO ActiveEnterprise standard wire format, which provides class information and packing rules for the TIBCO Adapter SDK set of data types. This format allows ActiveEnterprise components to perform extra validation on messages sent or received. This format can be used in conjunction with TIBCO Rendezvous.

- `aeXml` wire format can be used in conjunction with either TIBCO Rendezvous or TIBCO Enterprise Message Service.

Control Information

The wire formats are distinguished, in part, by the control information included when data is sent on the wire.

rvMsg

When a custom adapter sends data in `rvMsg` format, no control information is included.

aeRvMsg and aeXml

When a custom adapter sends data in `aeRvMsg` or `aeXml` format, the following control information is generated and set by the SDK:

- ^pfmt^—Present for backward compatibility.
- ^ver^—Version of the wire format.
- ^type^—Specifies how the data in the payload field is packed.
- ^prefixList^—List of strings for qualifying class names. This is generated and set by the SDK only if message contains a serialized `MData` or `MData` subclass.
- ^data^—Payload data.
- ^xmldata^—Payload data.
- ^tracking^—Always a unique identifier defined by the source. Set by the source and carried forward by intermediate components.

Table 31 lists the data type and value of each field for `aeRvMsg` and `aeXml`.

Table 31 aeRvMsg and aeXml Control Information

Name	aeRvMsg Type	aeRvMsg Field Value	aeXml Type	aeXml Field Value
^pfmt^	TIBRVMSG_INT(2)	10	string	10
^ver^	TIBRVMSG_INT(2)	1	string	1
^type^	TIBRVMSG_INT(2)	1	string	2
^prefixList^	TIBRVMSG_MSG	List of strings.	string	List of strings.
^data^	TIBRVMSG_MSG	Payload data.	N.A.	N.A.
^xmldata^	N.A.	N.A.	string	Payload data.
^tracking^	TIBRVMSG_MSG	Tracking identifier and any associated application information	string	Tracking identifier and any associated application information

Obsolete

The `^guid^` field is obsolete.

Advisory Handling

TIBCO Adapter SDK provides an advisory methodology for developers and end users to report or listen to advisory messages generated by TIBCO Adapters and TIBCO Rendezvous.

This feature is typically used in conjunction with existing TIBCO Rendezvous programs that use advisories.

Both TIBCO Rendezvous and the TIBCO Adapter SDK software present asynchronous advisory messages to custom adapters based on the SDK. Advisory messages indicate errors, warnings, and other information.

In contrast to status codes, which indicate success or failure *within* a specific TIBCO Adapter SDK call, asynchronous advisory messages notify custom adapters of events that occur *outside* of the adapter's direct flow of control. For example, the adapter is processing inbound messages too slowly, causing the TIBCO Rendezvous daemon's message queue to overflow.

Receiving Advisory Messages

Custom adapters can receive advisory messages in the same way as any other messages—by listening to subject names.

For example, the subject `_RV.*.SYSTEM.>` matches all advisories related to TIBCO Rendezvous communications. Programs can also listen for specific advisories. Where the advisory becomes available depends on whether they are related to a transport.

- Advisories related to a specific transport present on that transport. A custom adapter that creates several transports might need to listen for advisory messages on each transport.
- Advisories not related to a specific transport present on the intra-process transport.

Advisory messages wait for dispatch in the queue that the custom adapter designates when creating the listener.

To handle advisories, a custom adapter can instantiate `MAdvisory` directly and use the default advisory handling capabilities, or create a custom advisory listener.

Advisory Listeners

The advisory listener represents an event handler. The SDK allows you to use the default SDK advisory listener or create a custom advisory listener.

The event handler should be implemented by a custom adapter that needs to take specific action when an advisory event occurs.

There can only be one advisory listener in a custom adapter. A new one will override the default behavior.

Standard Advisory Listener

The SDK provides a default advisory listener that reports advisory messages being collected to the `MTrace` object. The advisory listener can listen to advisory messages from the sources listed in Table 32.

Table 32 Advisory Sources

Source	Description
TIBCO Rendezvous	Messages generated by TIBCO Rendezvous.
SDK	Messages from the SDK to help resolve any conflict during publication, subscription, or operation.
AE Operation Server	Advisory messages sent when an ActiveEnterprise operation server receives a request that it cannot serve. This could be because the incoming messages is in a bad format or cannot be demarshalled by the server.
TIBCO Administrator	Advisory messages are sent when there are changes in repository data.
User-defined	User-defined advisory. *Not Recommended* unless there is a special situation.

A typical usage scenario is to have an application listen to TIBCO Rendezvous error advisories such as `_RV.ERROR.>`. Using a simple configuration with the default advisory listener, any TIBCO Rendezvous error would be routed directly to the `errorRole` of the SDK `MTrace` object.

Custom Advisory Listener

You can replace the SDK default advisory listener with a custom advisory listener. For example, a custom advisory listener can be used to handle `_RV.RVCM.COLLISION`. In this case, the application might need to take specific action. To extend the default behavior, you can create a subclass that inherits `MAdvisoryListener` and explicitly calls the superclass callback.

To set a custom advisory listener, use the `MApp` method `setAdvisoryListener()`.

Advisory Publisher

The SDK provides an API to generate user-defined advisories. See the description of the `MAdvisory::send()` methods in the online API documentation.

The advisory publisher provides standard SDK advisory events for anomalies.

User advisories are intended for situations where there is no other way of notifying internal code that an unusual situation has occurred. For example, in cases where an API is using the SDK to offer services, it is possible that exceptions cannot be thrown when errors occur. It may be appropriate to send a user-generated advisory under these circumstances.

Advisory Subject Format

TIBCO Rendezvous Advisory Message

TIBCO Rendezvous advisory messages contain the following subject name as specified in the *TIBCO Rendezvous Concepts*:

```
_RV.<class>.<source>.<category>.<role>.<condition>.<name>
```

TIBCO Adapter SDK Advisory Message Structure

SDK advisory messages have the following structure:

```
    _SDK.<class>.<category>.<name>
```

or

```
    _SDK.<class>.<category>.<subject suffix>
```

Table 33 lists the message fields and descriptions.

Table 33 Advisory Message Fields

Field	Description
class	Severity of the situation. There are three classes of advisory messages: `ERROR`—problem is severe and not repairable. `WARN`—problem exists during the operation but the program managed to resolve the problem. Undesirable side effects might occur. `INFO`—information that may be of interest to the user. SDK takes no action. It only detects these conditions and reports to the users.
category	The following categories are available: `RPC`—operations-related advisory messages `USER`—user-reported advisory messages

Table 33 Advisory Message Fields (Cont'd)

Field	Description
subject suffix	Advisory related to a subject name on a publisher, subscriber, or an operation. This can be a string with arbitrary information.
name	Advisory related to a component name such as a publisher, subscriber, or a session.

Advisory Message Format

Advisory messages are free-form. They are attached to an `MTree` to be sent out. All SDK advisory messages are wrapped into an `MTree` and sent to the advisory event listener. The advisory event mechanism delivers messages among the threads of a program and does not go into the network.

Here are the predefined SDK advisory messages:

- `_SDK.ERROR.RPC.BADMSGFORMAT`

  Advisory messages generated when an ActiveEnterprise Operation server receives an unknown request.

  The original message is appended as a part of the advisory message, but the destination name is not included. You should be able to determine this from the operation server endpoint itself. You can then use this information to acknowledge the message manually, in case of durable subscribers with auto acknowledge disabled.

- `_SDK.ERROR.RPC.BADOPERATIONCLASS`

  Advisory messages generated when an ActiveEnterprise Operation server receives a legal request but cannot deserialize the request parameters.

- `_SDK.ERROR.HAWK.BADMSGFORMAT`

  (C++ only) Advisory message generated when a TIBCO Hawk microagent receives an unknown or malformed request. The advisory could be caused by Adapter SDK subscribing to the unsupported Hawk subject `_HAWK.AMI.DISCOVERY` when the Hawk standard microagent is enabled.

- `_SDK.WARN.HAWK.MISSING_METHOD`

  (C++ only) Advisory message generated when a TIBCO Hawk method implementation is missing from the Hawk microagent for an incoming request.

- `_SDK.ERROR.JMS.RECEIVE_FAILED`

  This advisory is sent to adapters with JMS subscribers that failed to receive messages due to a JMS internal error. This can occur if the JMS server crashes or terminates during a receiving operation: `jms status = SERVER_NOT_CONNECTED`. This error advisory is an unrecoverable condition. Applications that rely on this JMS session must exit.

However, if the JMS session has a fault tolerant connection URL specified, JMS will fail over to next server. If the JMS session has a reconnect URL specified instead, this advisory occurs only after JMS has exhausted all its reconnection attempts. Note that no error or exceptions are thrown by JMS while it is attempting to reconnect.

User-Defined Advisories

Use the `MAdvisory::send()` method for user-defined advisories.

C++ Code Example

```
MAdvisory * pAdvisory = pMApp->getAdvisory();
pAdvisory->send( "ADVISORIES.DB.SEVERE"
                 ,"ERROR"
                 ,"Insert operation failed
                 ,"queued for retry"
                 ,"remoteLoggingSession" );
```

Java Code Example

In Java, use the `MAdvisory.send()` method, as in the following example:

```
MAdvisory advisory = mapp.getAdvisory();
advisory.send("ADVISORIES.DB.SEVERE",
              "ERROR",
              "Insert operation failed",
              "remoteLoggingSession");
```

Using the MPlugin Class

The `MPlugin` class allows applications to add extra functionality to an adapter at runtime without recompiling the adapter. Possible uses of the `MPlugin` class are:

- userExit plug-in
- loading new `MApp` components

See the *SDK_HOME*`/examples/java` directory for a plug-in example.

Defining a Plug-in

To define a plug-in, you need to create a subclass of `MPlugin` and implement one C function and four C++ methods.

Table 34 Function and Method for a Plug-in

Function & Method	Description
`CreateNewPlugin()` C function	Calls the plug-in constructor with a C interface.
`MPlugin::MPlugin()` method	Plug in constructor.
`MPlugin::~MPlugin()` method	Plug in destructor.
`MPlugin::onInitialization()` method	Initializes the plug in.
`MPlugin::onTermination()` method	Terminates the plug in.

Configuring a Plug-in

You can configure a plug-in by using the TIBCO Designer to add a custom plug-in object to the adapter instance description stored in the project repository.

For example, to configure a Java plug-in `MyPlugin.java`:

1. Click the **Edit Adapter XML** button in the Configuration tab of Generic Adapter Configuration resource.
2. Add the following entries:

```
AESDK:plugins
AESDK:plugin
AESDK:name pluginObject
AESDK:className MyPlugin
AESDK:verbose true
```

```
AESDK:pluginObject true
AESDK:filename MyPlugin.java
```

Where `MyPlugin` and `MyPlugin.java` specify the class name and file name of the sample custom plug-in.

3. Click **OK** to close the XML file.
4. Click **Apply**.

Running an Adapter with a Plug-in

To load a plug-in into the adapter, specify the `system:plugin` command-line argument as follows:

`"-system:plugin` *your_plugin_shared_library*`"`

Several plug-ins can be loaded by a single adapter.

Before using this feature, be sure to call `MAppProperties::setCommandLine()`.

Transformation Plug-in

The transformation plug-in allows developers to transform inbound and/or outbound messages for any adapter built with the Adapter SDK. It could be used, for example, for performing localized transformations using XSLT.

In TIBCO Designer, a transformation plug in is called a Message Filter.

Usage Scenarios

- The adapter wants to transform all outbound XML with XSLT

  The application to which the adapter connects understands XML natively. The adapter code retrieves messages and makes use of the `MInstance::toXML()` methods. The corresponding XSD is available in-memory and on disk if you have exported it. To achieve local transformations to a *canonical format*, XSLT is the logical choice. These transformations can be applied to all outbound endpoints. Inbound transformations most likely don't make sense.

- Arbitrary custom data manipulation is desired (not XSLT)

  Some custom code is required to manipulate messages before sending them to a third-party application. For example, a legacy TIBCO Rendezvous application expects its data in a particular format. The only solution is to write an intervening adapter between packaged adapters and the TIBCO Rendezvous application to do TIBCO Rendezvous transformations, or to use other TIBCO products (for example, TIBCO ActiveMatrix BusinessWorks).

Implementation

The low-level callout behavior is made available by an `MTransformationPlugin` class that is part of the API (both C++ and Java). See the reference documentation for more information.

The low-level callout is specified in the TIBCO Designer:

1. Select the adapter configuration resource and choose the Configuration tab.
2. Click the **Browse** button next to the Message Filter field to select the class that implements the transformation plug-in.

The transformation plug-ins are applied *only* to `MSubscriber`, `MPublisher`, `MRpcClient`, and `MRpcServer` endpoints.

You can either throw an exception or return `FALSE` to stop message flow in the `transformInbound()` or `transformOutbound()` method.

If the `transformOutbound()` method throws an exception, it will be propagated back to user.

If the `transformInbound()` method throws an exception, an advisory will be sent out on the subject, `_SDK.ERROR.PLUGIN.PLUGIN_TRANSFORMATION_ERROR`.

When using the `transformInbound()` method in async mode, the message flow does not stop. An advisory message is sent on the above subject.

Example

The following example illustrates how to use the low-level plug-in.

Not all details are present here (for example, the `DLL_EXPORT` macro details are omitted).

```
// GOES IN THE HEADER FILE
extern “C”
{
   DLL_EXPORT MException *
   CreateTransformationPlugin( MApp * pMApp
,MTransformationPlugin * & pTPlugin );
   DLL_EXPORT void
   DestroyTransformationPlugin( MTransformationPlugin * & pTPlugin );
}

class CustomPlugin : public MTransformationPlugin
{
   CustomPlugin( MApp * pMApp,
   MString const & pluginName,
   MString const & pluginID ) throw (MException);

   virtual Mboolean
   transformOutbound( MComponent const & endpoint, MTree & outboundTree )
                      throw (MException);

   virtual Mboolean
   transformInbound( MComponent const & endpoint, MTree & inboundTree ) throw
(MException);
```

```
// GOES INTO THE CPP FILE
CreateTransformationPlugin( MApp * pMApp,
   MTransformationPlugin * & pTPlugin )
{
   try {
CustomPlugin * pCP =
new CustomPlugin( pMApp, “MyTransformer”, “Converter v1.0” );
if pCP == NULL
return new MException(…);
   catch( MException e ) {
return new MException(…);
   }
   return NULL;
}

void DestroyTransformationPlugin( MTransformationPlugin * pTPlugin )
{
   delete pTPlugin;
}


CustomPlugin::CustomPlugin( MApp * pMApp,
   MString const & pluginName,
   MString const & pluginID ) throw (MException)
   :MTransformationPlugin( pMApp, pluginName, pluginID )
{ … }

virtual Mboolean
   CustomPlugin::transformOutbound(MComponent const & endpoint, MTree &
outboundTree ) throw (MException)
{
return veilMTree( outboundTree );
}

virtual Mboolean
CustomPlugin::transformInbound(MComponent const & endpoint, MTree & inboundTree )
throw (MException)
{
return unveilMTree( inboundTree );
}
```

Subject Names

Each TIBCO Adapter SDK message bears a *subject* name.

Data-producing programs generate new data messages, label them with subject names, and send them using TIBCO Adapter SDK software. Data consumers receive data by listening to subject names. A consumer listening to a name receives all data labeled with a matching name, from the time it begins to listen until it stops listening.

Subject Name Syntax

Subject-based addressing™ technology places few restrictions on the syntax and interpretation of subject names. System designers and developers have the freedom (and responsibility) to establish conventions for using subject names. The best subject names reflect the structure of data in the application itself.

Each subject name is a string of characters that is divided into elements by the dot (.) character. It is invalid to incorporate the dot character into an element by using an escape sequence.

TIBCO Adapter SDK limits subject names to a total length of 255 characters (including dot separators). The maximum element length is 252 characters (dot separators are not included in element length). Typical subject names are shorter and use fewer elements. To maximize speed and throughput rates, use short subject names.

The following are examples of *correct* subject names:

- `NEWS.LOCAL.POLITICS.CITY_COUNCIL`
- `NEWS.NATIONAL.ARTS.MOVIES.REVIEWS`
- `CHAT.MRKTG.NEW_PRODUCTS`
- `CHAT.DEVELOPMENT.BIG_PROJECT.DESIGN`
- `News.Sports.Baseball`
- `finance`
- `This.long.subject_name.is.valid.even.though.quite.uninformative`

The following are examples of *incorrect* subject names:

- `News..Natural_Disasters.Flood` (null element)
- `WRONG.` (null element)
- `.TRIPLE.WRONG..` (three null elements)

Table 35 lists special characters in subject names.

Table 35 Special Characters in Subject Names

Character	Character Name	Special Meaning
_	Underscore	⚠ Subject names beginning with underscore are reserved. It is illegal for application programs to send to subjects with underscore as the first character of the first element, except `_INBOX` and `_LOCAL`. It is legal to use underscore elsewhere in subject names.
.	Dot	Separates elements within a subject name.
>	Greater-than	Wildcard character, matches one or more trailing elements.
*	Asterisk	Wildcard character, matches one element.

It is recommended that you not use tabs, spaces, or any unprintable character in a subject name.

Using Wildcards to Receive Related Subjects

Programs can listen for wildcard subject names to access a collection of related data through a single subscription.

- The asterisk (`*`) is a wildcard character that matches any one element. The asterisk substitutes for whole elements only, not for partial substrings of characters within an element.
- Greater-than (`>`) is a wildcard character that matches all the elements remaining to the right.

A listener for a wildcard subject name receives any message whose subject name matches the wildcard.

The examples in Table 36 illustrate wildcard syntax and matching semantics.

Table 36 Using Wildcards to Receive Related Subjects

Listening to this wildcard name	Matches messages with names like these:	But does not match messages with names like these (reason):
`RUN.*`	`RUN.AWAY` `RUN.away`	`RUN.Run.run` (extra element) `Run.away` (case) `RUN` (missing element)
`Yankees.vs.*`	`Yankees.vs.Red_Sox` `Yankees.vs.Orioles`	`Giants.vs.Yankees` (position) `Yankees.beat.Sox` (vs≠beat) `Yankees.vs` (missing element)

`*.your.*`	`Amaze.your.friends`
	`Raise.your.salary`
	`Darn.your.socks`
		`your` (missing elements)
		`Pick.up.your.foot` (position)

`RUN.>`	`RUN.DMC`
	`RUN.RUN.RUN`
	`RUN.SWIM.BIKE.SKATE`
		`HOME.RUN` (position)
		`Run.away` (case)
		`RUN` (missing element)

Table 37 shows subject names that use invalid wildcards.

Table 37 Invalid Wildcards in Subject Names

Invalid Wildcards	Reason
`abc*xyz`	Asterisk (*) must take the place of one whole element, not a substring within a element.
`Foo.>.baz`	Greater-than (>) can only appear as the right-most character. TIBCO Adapter SDK software interprets this as a specific subject name.

⚠

It is not recommended to send messages to wildcard subject names.

Although transports do not prevent you from sending to wildcard subjects, doing so can trigger unexpected behavior in other programs that share the network.

It is invalid for certified delivery transports to send to wildcard subjects.

Distinguished Subject Names

Names that begin with an underscore character (_) are called *distinguished subject names*. Distinguished names indicate special meaning, special handling, or restricted use.

Table 38 Subject Names with Special Meanings

Prefix	Description
`_INBOX.`	All inbox names begin with this prefix. Programs may not create inbox names except with inbox creation calls. Programs must treat inbox names as opaque, not modify them, and refrain from making inferences based on the form of inbox names.
`_LOCAL.`	Messages with subject names that have this prefix are only visible and distributed to transports connected to the same TIBCO Adapter SDK daemon as the sender. For example, a program listening to the subject `_LOCAL.A.B.C` receives all messages sent on subject `_LOCAL.A.B.C` from *any* transport connected to the *same* daemon. A TIBCO Adapter SDK daemon does not transmit messages with `_LOCAL` subjects beyond that daemon.
`_RV.`	Subject names with this prefix indicate advisory messages, including informational messages, warnings and errors. Programs must not send to subjects with this prefix.
`_RVCM.`	Subject names with this prefix indicate internal administrative messages associated with certified message delivery. Programs must not send to subjects with this prefix.
`_RVCMQ.`	Subject names with this prefix indicate internal administrative messages associated with distributed certified delivery transports. Programs must not send to subjects with this prefix.
`_RVFT.`	Subject names with this prefix indicate internal administrative messages associated with TIBCO Adapter SDK fault tolerance software. Programs must not send to subjects with this prefix.
`_RVDS.`	Subject names with this prefix indicate protocol messages associated with TIBCO Adapter SDK DataSecurity software (sold separately). Programs must not send to subjects with this prefix.

Chapter 9 TIBCO Adapters and TIBCO Hawk

This chapter discusses integrating SDK-based adapters with TIBCO Hawk.

Topics

TIBCO Adapter SDK and TIBCO Hawk

This section provides an overview of TIBCO Hawk and an overview of the SDK integration capabilities.

TIBCO Hawk Overview

TIBCO Hawk is a tool for enterprise-wide monitoring and managing of distributed applications and systems. System administrators can use it to monitor nodes in a wide area network of any size.

TIBCO Hawk can be configured to monitor system and application parameters and to take actions when predefined conditions occur. These actions include:

- sending alarms that are graphically displayed in the TIBCO Hawk display
- sending e-mail
- paging
- running executable
- modifying the behavior of a managed application

Unlike other monitoring applications, TIBCO Hawk relies on a purely distributed intelligent agent architecture using publish/subscribe to distribute alerts.

TIBCO Hawk uses TIBCO Rendezvous for all messaging and thus gains the benefits and scalability from the TIBCO Rendezvous features of publish/subscribe, subject name addressing, interest-based routing, and reliable multicast.

TIBCO Hawk is a purely event-based system that uses alerts. The agents are configured with rules that instruct them from what and how to monitor to what actions to take when problems are detected. Thus the workload is fully distributed throughout the enterprise. Every agent is autonomous in that it does not depend on other components to perform its functions.

The TIBCO Hawk Enterprise Monitor consists of the following components:

Display GUI front end that displays alarms and provides editors to create rule bases, create tests, view messages, and invoke microagents to request information or initiate an action.

Agents Intelligent processes that perform monitoring and take actions as defined in rules.

Rulebases Rules that are loaded by agents to determine agent behavior.

Application Management Interface (AMI) Manages network applications through TIBCO Rendezvous and supports communication between a network application and monitoring TIBCO Hawk agents, including the ability to examine application variables, invoke methods, and monitor system performance.

Microagents Feed information back to TIBCO Hawk and expose action methods to rulebases.

For more information, see the TIBCO Hawk documentation.

TIBCO Hawk and Adapter Applications

With the TIBCO Adapter SDK and TIBCO Hawk, you can do the following:

- Use generally available functionality, that is, manage your adapter through processes, log files, and command line executables.
- Use available TIBCO Hawk methods. For example, the `_describe` and `_heartbeat` methods are handled automatically.
- Use the TIBCO Hawk AMI methods in the microagent provided by the SDK. These methods know how to do a lot of the monitoring activities commonly required by adapter applications. See Predefined TIBCO Hawk Microagent Methods on page 156.
- Use the TIBCO Adapter SDK for the TIBCO Hawk AMI to define your own methods that can be invoked from the TIBCO Hawk display. See TIBCO Adapter SDK API to TIBCO Hawk on page 159.

Predefined TIBCO Hawk Microagent Methods

This section discusses methods that are predefined as part of the SDK. These methods facilitate monitoring and debugging a custom adapter.

For complete documentation of each microagent method, see Appendix C, TIBCO Adapter SDK Hawk Microagents and Methods.

Terminology

To understand the following sections, you need to be familiar with some basic concepts and terms used by TIBCO Hawk:

- The *Hawk Application Management Interface (AMI)* is an agreement on procedures and TIBCO Rendezvous message formats used to instrument an application. Selected methods of an application (in this case, a custom adapter) are made accessible to a TIBCO Hawk agent for the purpose of monitoring and managing the application.
- A *microagent* is used by TIBCO Hawk to collect information and carry out tasks, usually relating to one type of managed resource or managed application. On the TIBCO Hawk display, each microagent is one menu item. TIBCO Hawk end users select first the microagent, then the desired AMI method within that microagent.

Microagents Provided by the SDK

The SDK creates two microagents. The associated sets of methods can then be used to monitor adapters.

- `COM.TIBCO.ADAPTER`. This microagent allows you to perform queries on all running adapters, regardless of their class/application.

  If you choose this item on the TIBCO Hawk display and invoke one of the associated AMI methods, the corresponding method is invoked in all running adapters.

  Generally, `COM.TIBCO.ADAPTER:n` and `COM.TIBCO.ADAPTER.xyz:n` behave the same except if you invoke a method on `COM.TIBCO.ADAPTER:n` through network query in TIBCO Hawk display, `COM.TIBCO.ADAPTER:n` causes it to perform queries on all running adapters.
- `COM.TIBCO.ADAPTER.`*xyz* (where *xyz* is the name of a class of adapters and could stands for the source or target software package with which the adapter

is interfacing). This microagent allows performing queries on one class of adapter.

If you choose this item on the TIBCO Hawk display and invoke one of the associated AMI methods, the corresponding method is invoked only in instances of the *xyz* adapter.

Both microagents implement the same methods. The functionality provided by these methods includes:

- RVCM ledger file monitoring and preregistration
- RV and RVCM subscribers/publishers discovery
- General adapter state and configuration information
- Tracing profile modification

By default, each instance of `MApp` has its microagent. Adapters can override this behavior in the Monitoring tab of the TIBCO Designer software.

Configuring the TIBCO Hawk Microagents

TIBCO Hawk microagents are configured using TIBCO Designer.

- To configure the default microagent, use the Monitoring tab of the adapter you are configuring.
- To configure application-specific microagents, you must edit the XML file. Select the Generic Adapter Configuration and click the **Edit Adapter XML** button.

See *TIBCO Designer Palette Reference* for detailed information on the fields you need to configure. You can access the book from TIBCO Designer using **Help > Designer Help**.

TIBCO Adapter SDK API to TIBCO Hawk

In addition to working with the predefined TIBCO Hawk AMI, the TIBCO Adapter SDK provides its own API to create AMI methods for use with TIBCO Hawk. The API is modified in several ways:

- **Data formats**—By default, data that are managed by TIBCO Hawk are TIBCO Rendezvous messages. However, SDK-based adapters can also send information in `MTree` format to TIBCO Hawk and retrieve information in `MTree` format from TIBCO Hawk.
- **Configuration information**—By default, an application that wants to use TIBCO Hawk must create a TIBCO Rendezvous session. If you use the SDK, `MApp` will establish a TIBCO Rendezvous session for use with TIBCO Hawk, based on the adapter configuration in the project repository.
- **Session management—**The SDK creates a TIBCO Hawk microagent (unless explicitly prohibited during adapter configuration) and registers any `MHawkMethod` instances as TIBCO Hawk methods. You can then invoke these methods from TIBCO Hawk to monitor the adapter.

TIBCO Hawk Integration Classes

The SDK makes the following classes available for TIBCO Hawk integration:

- `MHawkMicroagent`—Encapsulates information about the `MHawkMicroagent`.

  During initialization, `MApp` creates an instance of this class unless the adapter instance description object specifies that a TIBCO Hawk microagent should not be available. Note that a microagent is available by default.

- `MHawkMethod`—Encapsulates a method.

  For each method you want to have available from TIBCO Hawk, create a subclass of `MHawkMethod` and implement its `processMethodInvocation()` method.

Creating User-Defined TIBCO Hawk Methods

This section explains how to create TIBCO Hawk microagents for C++ adapters and Java adapters.

Creating TIBCO Hawk Methods in C++

To create a method for a C++ adapter:

1. Use the TIBCO Designer software to specify information about the method name and method input and output parameters in the adapter instance description object. Select the adapter, then click **Edit Adapter XML** to define this information. The following association attributes describe TIBCO Hawk methods.

   — hawk association attribute

   — hawk.method association attribute

   — hawk.method.inputParameter association attribute

   — hawk.method.outputParameter association attribute

   For an example, see *TIBCO Designer Palette Reference*, available through **Help > Designer Help** from TIBCO Designer.

2. In the custom adapters, create a subclass of `MHawkMethod` and define the appropriate methods, and the destructor. Note that all methods must have been defined in TIBCO Designer for that adapter instance.

   The following class is from the `testHawk` example:

```
class Hawk_GetDebugValues : public MHawkMethod
{
public:
 Hawk_GetDebugValues( MHawkMicroAgent& refHawkMicroAgent )
  : MHawkMethod( "getDebugValues", refHawkMicroAgent ),
 {}
```

3. Implement the `MHawkMethod::processMethodInvocation()` method in the subclass. For the subclass above, the method is implemented as follows:

```
void processMethodInvocation(const MTree& inMTree,
                      MTree& outMTree) throw(MException)
{
   // Ignore the mtree passed.
   int level = m_pTestHawkDebug->getDebugLevel();
   Mboolean bStatus = m_pTestHawkDebug->getDebugOnStatus();
   MString sMessage = m_pTestHawkDebug->getDebugMessage();
```

```
       outMTree.append("Level", level);
       outMTree.append("On", bStatus?1:0);
       outMTree.append("Message", sMessage.c_str());
       return;
    }
```

4. Instantiate the method and associate it with the microagent in the `onInitialization()` method:

```
Hawk_GetDebugValues *  pHawkMethod =
               new Hawk_GetDebugValues(someHawkMicroAgent);
if ( pHawkMethod != NULL ) {
     pHawkSession->addMethod( pHawkMethod );
```

Creating TIBCO Hawk Methods in Java

Creating user-defined TIBCO Hawk methods with the Java SDK differs slightly from the process used for the C++ SDK.

To create a method:

1. Use the TIBCO Designer software to specify information about the method name and method input and output parameters in the adapter instance description object. Select the adapter, then click **Edit Adapter XML** to define this information. The following association attributes describe TIBCO Hawk methods.
   — hawk association attribute
   — hawk.method association attribute
   — hawk.method.inputParameter association attribute
   — hawk.method.outputParameter association attribute

For an example, see *TIBCO Designer Palette Reference*, available through **Help > Designer Help** from TIBCO Designer.

2. In the Java adapter application, create an object and the method you want to have invoked from TIBCO Hawk. The method must match a method defined in the adapter instance description object as follows:
   — Same method name
   — Same name and type of parameters
   — Same return type. (Note that if multiple values are returned, the return type must be `MTree`.)
3. Get the `MHawkMicroagent` instance from the `MApp MHawkRegistry` and call `MHawkMicroagent.setMonitoredObject()`, providing the object that contains the method you defined as the argument.

Through introspection, the SDK finds methods at runtime that match those defined in the adapter instance description object and make them available from TIBCO Hawk.

For instructions on how to choose the input and output parameters, see the class description for `MHawkMethod` in *TIBCO Adapter SDK Java API*.

Chapter 10 Getting Started: Hello World Adapter

This chapter explains how to configure and write a simple "Hello World" adapter. For detailed information on adapter configuration and implementation, see Chapter 11, Custom Adapter Example: zapadapter.

Topics

Prerequisites

Before starting, make sure that you have the required software available on your machine. See the *TIBCO Runtime Agent Installation Guide* for more information.

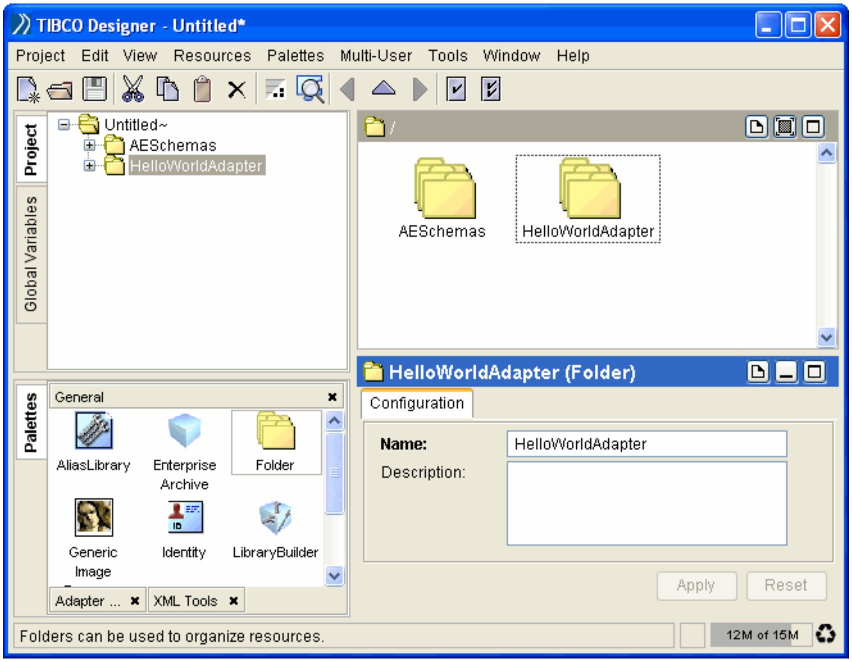
Preparing the Adapter Configuration

The TIBCO Designer software allows you to define configuration data (and optionally metadata) for your custom adapter. To create an adapter instance that holds the minimum configuration information, follow these steps:

1. Start TIBCO Designer with a new empty project.
2. Click **Cancel** when you are prompted for Save Project information. You can save the project later.

   For more information on answering the prompts given at startup, see the *TIBCO Designer User's Guide* available through **Help> Designer Help**.
3. Click **Palettes> General > General** to display the general palette that contains the folder resource.
4. From the General palette, drag a folder to the design palette and name it `HelloWorldAdapter`. Click the **Apply** button.

Figure 23 Create the HelloWorldAdapter



Name: HelloWorldAdapter

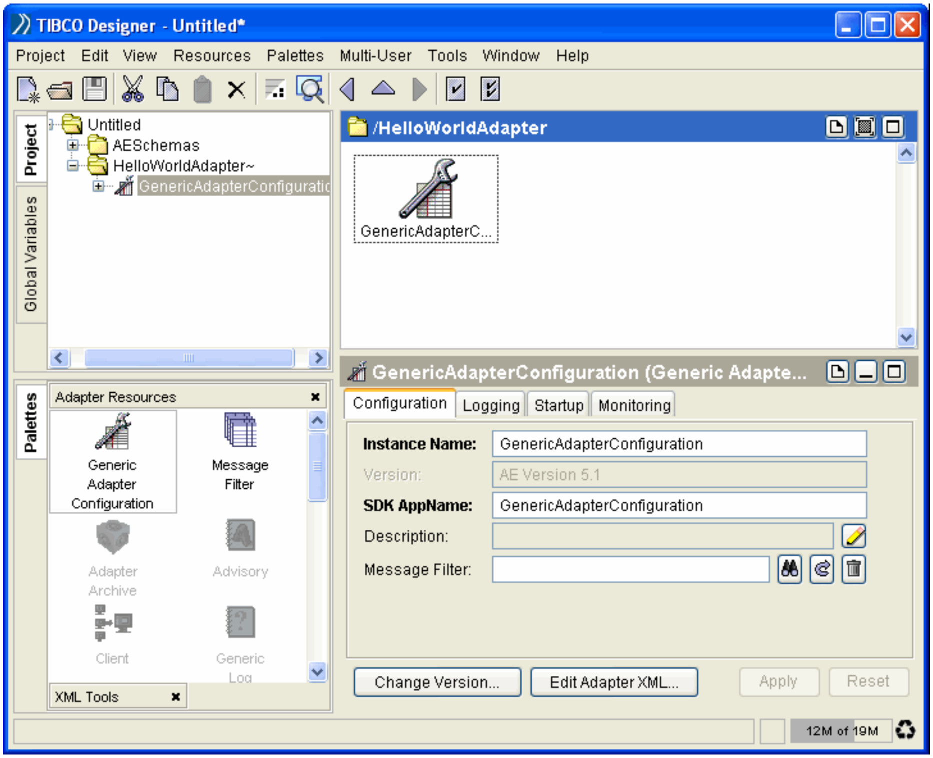
Description:

Folders can be used to organize resources.

5. Click **Palettes > Adapters > Adapter Resources** to display the adapter resources palette.

6. In the upper left project panel, double-click the `HelloWorldAdapter` folder to open it, then drag the `Generic Adapter Configuration` resource into the design panel.

Figure 24 Add Generic Adapter Configuration Resource



Instance Name: GenericAdapterConfiguration
Version: AE Version 5.1
SDK AppName: GenericAdapterConfiguration
Description:
Message Filter:

7. In the Instance Name field, type **`defaultInstance`**.

8. In the SDK AppName field, type **`helloWorld`**. This is the short name of the adapter that is used in the source code. Click **Apply**.

   You have now specified the minimum configuration information for an adapter.

This example does not configure any services, sessions, or endpoints. They are discussed in the second example, Custom Adapter Example: zapadapter on page 173.

9. Click **Project > Save** to save the configuration.

10. Click **Project > Export Full Project**. In the dialog box, click the **Local Repository** tab, then provide the following information:
    a. In the Project Name field, type **`HelloWorldRepoOne`**.
    b. In the Dir Name field, click **Browse** and select the directory to save the project. Click **OK**.
    c. Click **OK** to save the project to the project repository.
11. Click **Project > Exit**.

A sample copy of the resulting repository can be found in your installation area under the `examples` directory. For example, on Windows: *SDK_HOME*`\examples\helloWorld`.

The Adapter Program

After the configuration is completed, the code can access the configuration file. This section lists the code for the Hello World Adapter in C++ and Java.

Hello World Code in C++

Sample code for the Hello World program is included as part of the SDK installation. To view the code, open the project file *SDK_HOME*`\examples\helloWorld\helloWorld.dsp` on Microsoft Windows platforms or the `helloWorld.cpp` file on UNIX platforms.

The adapter application manager class is defined as follows:

```
#include <Maverick.h>

#include <iostream.h>
class HelloWorldApp : public MApp

{
  public:
    HelloWorldApp( MAppProperties* pMAppProperties)    //constructor
    :MApp(pMAppProperties, NULL) {}

    virtual ~HelloWorldApp() {}                        //destructor


    void onInitialization() throw (MException)         //you must define this method
                                                     //it is called by MApp::start()
    {
      cout << " Hello World Adapter " << endl ;
    }
    void onTermination() throw (MException) {}       //you must define this method
                                                     //it is called by MApp::stop()

};
```

`main()` is defined as follows:

1. It first creates an `MAppPropterties` instance to specify application properties.

```
int main(int argc, char* argv[])
{

   MAppProperties appProperties;

   appProperties.set(MAppProperties::APPVERSION, "1.0");
   appProperties.set(MAppProperties::APPINFO, "TIBCO Adapter Hello World Example");
   appProperties.set(MAppProperties::APPNAME, "helloWorld");
   // set the repository location
   appProperties.set(MAppProperties::REPOURL, "HelloWorldRepo.dat");

   appProperties.set(MAppProperties::CONFIGURL,
                     "/tibco/private/adapter/HelloWorldAdapter/defaultInstance");

   // store the command line args with MApp
   appProperties.setCommandLine(argc, argv);
```

2. The program then creates the `MApp` application manager, passing in the properties just defined.

```
   try {
      HelloWorldApp* pHelloWorldApp =
         new HelloWorldApp(&appProperties);
```

3. Next comes a call to the `start()` method, which starts the event loop by default. For this example, there is no need to start the event loop, so pass in `false`.
4. The `stop()` method is invoked right after that. While the application did not enter the event loop, the `stop()` method must be called because it performs any internal cleanup and will then call `onTermination()` to perform developer-defined cleanup.

```
      pHelloWorldApp->start(Mfalse);
      pHelloWorldApp->stop();

      delete pHelloWorldApp;
   }
   catch(MException& e )
   {
      cout << "Exception caught in main:
                         "<< e.getType() << ":" << e.getDescription() << endl;
   }

   return 0;

} // main
```

You can compile and link the program using:

- Visual C++ on Microsoft Windows platforms
- `make` on Unix platforms

Hello World Code in Java

The `helloWorld` example file is included in the installation media as:
SDK_HOME`\examples\helloWorld\helloWorld.java`

The `MAppProperties` class is defined as follows:

```
Define the helloWorld class as follows:
public class helloWorld
{
  MApp app;

public helloWorld( String[] args ) throws Exception
   {
      // Create the Mapp
      // you can also try to pass in parameter as -system:configurl and
      //-system:repourl
        MAppProperties p = new MAppProperties(
            "helloWorld",
            "1.0",
            "Hello World Adapter",
            "/tibco/private/adapter/HelloWorldAdapter/defaultInstance",
            "HelloWorldRepo.dat",
            args);
      System.out.println ("repo = "+p.getRepoURL());
      System.out.println ("config = "+p.getConfigURL());

        app = new helloWorldApp( p );
        app.start(false);
        app.stop();
   }
```

```
import java.util.*;
import com.tibco.sdk.*;

class helloWorldApp extends MApp                   //create application manager

{
  public helloWorldApp( MAppProperties p )
  {
    super( p );
  }

  /** Hook to perform application-specific behavior during
  * initialization
  */
  protected void onInitialization() throws MException          //required
  {
    System.out.println (" Hello World Adapter ");
  }

  /** Hook to perform application-specific behavior during
  * shut-down
  */
  protected void onTermination() throws MException
  {
  }

} // end of class helloWorldApp



public static void main( String[] args ) {
  try {
    new helloWorld( args );
  }
  catch( Exception fatal ) { fatal.printStackTrace(); }
  }
} // end of class helloWorld
```

---

1. Set the environment variable CLASSPATH to include all dependent JAR files.
2. Compile using the following command.

   ```
   javac helloWorld.java
   ```

3. Run the `helloWorld` adapter using the following command.

   ```
   java helloWorld
   ```

Chapter 11 Custom Adapter Example: zapadapter

This chapter explores an example program that illustrates how to build a simple custom adapter.

Topics

Overview

This chapter first explains how to make some design decisions, then explains the implementation of those decisions by presenting code fragments from an example adapter program, `ZapAdapter` (`ZAP` is a fictitious ERP system).

All code fragments in this chapter are taken from the `zapadapter` example program.

The example code in this chapter is C++, but the concepts are essentially the same for both APIs. Java programmers can find the `zapadapter` example included in the examples folder.

The *SDK_HOME*`/resourceKit/deployableAdapter` directory contains a ZAP adapter example program that was modified to run in an administration domain. The directory also contains step-by-step instructions on how to change ZAP adapter to make it compatible with TIBCO Administrator.

Analysis and Design

Before implementing the adapter, you must be clear about the problem to be solved. This section illustrates how an example problem statement can be mapped to a design.

Problem Statement

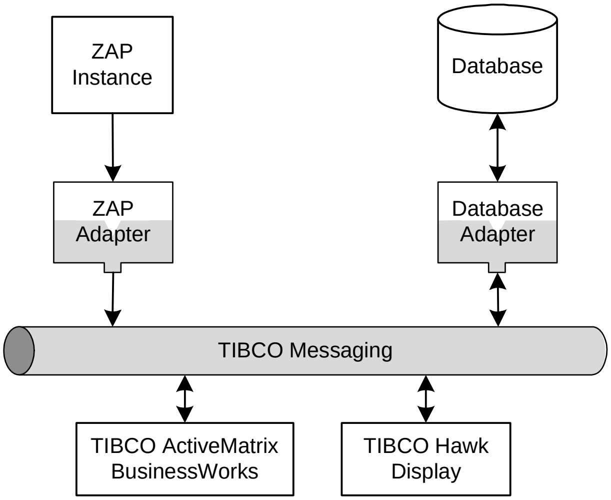
The `ZapAdapter` application retrieves data from a `ZAP` source application and publishes it to the messaging system. `ZAP` has the following characteristics:

- `ZAP` is an enterprise order processing application.
- `ZAP` provides a `DataMapper` (in C++ only) and `ZAPAppData` classes to access `ZAP` data.
- `ZAP` does not provide event-driven APIs.

Because the `ZAP` system's data access API does not support an event-driven approach to retrieving data from the `ZAP` system, `ZapAdapter` checks every *n* seconds for new events. If there are unprocessed events, the data associated with the events must be published.

Figure 25 shows how this example fits into the TIBCO messaging application framework.

Figure 25 ZAPAdapter and the TIBCO Enterprise Model



- An instance of ZAP communicates with TIBCO messaging through ZapAdapter.
- The adapter sends the data to a listening adapter. The database, which is always kept current by the actions of `ZAP` adapter, is used for queries. This design offloads query processing from the `ZAP` instance.
- Both adapters, as well as other TIBCO ActiveEnterprise applications, are monitored through TIBCO Hawk.
- TIBCO ActiveMatrix BusinessWorks could be set up to perform data conversion. This allows users of the database to work with queries they are familiar with.

Elements of Implementation

The implementation of `ZapAdapter` must provide the following elements:

- **Initialization and startup**. In the example, the adapter needs to:
  - Establish a connection with the `ZAP` instance.
  - Create a publisher and a timer event sources.
- **Event sources.** The application must include the following event source:
  - Timer event source to wait for timer events and poll for data.
- **Publisher**. Publishes the data retrieved by the timer event source.
- **Event listener subclass** with `onEvent()` methods.

  In this example, the Timer event handler polls data when informed by timer event source.
- **Data conversion element**. In the `ZAP` example, data conversion is done explicitly by a `ZapEventListener`, `DataMapper` (in C++ only) and `ZAPAppData` class. The `ZapEventListener` class accepts data from the `ZAP` instance and converts it to `MInstance` format. Specifically, this example retrieves two tables, `Customer` and `Contract` that share an ID field.
- **Configuration.** The configuration is performed in TIBCO Designer. See Specifying Configuration Information on page 178.

Components of ZapAdapter Sample Application

This section gives an overview of the `ZapAdapter` files and their content.

Table 39 ZapAdapter Files

File	Function/Class/Method	Description

`ZapAppData.h`
`ZapAppData.cpp`
	`getZAPContract()`
	`getZAPCustomer()`
	`getPendingCustomerEvent()`
	`addCustomer()`
	`addContract()`
		External library used by the `ZAP` adapter to retrieve and update data.

`ZapAdapter.h` `ZapAdapter.cpp`	`ZapAdapter class main()`	`MApp` subclass with `onInitialization()` and `onTermination()` method. Creates an event handler and publisher instances. Top-level control function `main()`.
`ZapEventListener.h` `ZapEventListener.cpp`	`TimerEventHandler` `SignalEventHandler`	Subclasses of `MEventListener` with `onEvent()` methods.
`ZAPConnection.h`	`ZapConnection` class with methods: `connect()`, `disconnect()`, and `getPendingCustomerEvent()`	Maintains ZAP connectivity logic. Connection parameters are passed through the properties object. This class creates and maintains the handle to the `ZAP` instance.
`DataMapper.h`	Data Mapper class	Maps `AccountInfo` and `OrderInfo` from `MInstance` to C++ and back.

Specifying Configuration Information

Before running any adapter application, you need to specify its configuration information. Configuration information and metadata information are specified using the TIBCO Designer software.

Creating a configuration for `zapAdapter` consists of the following tasks:

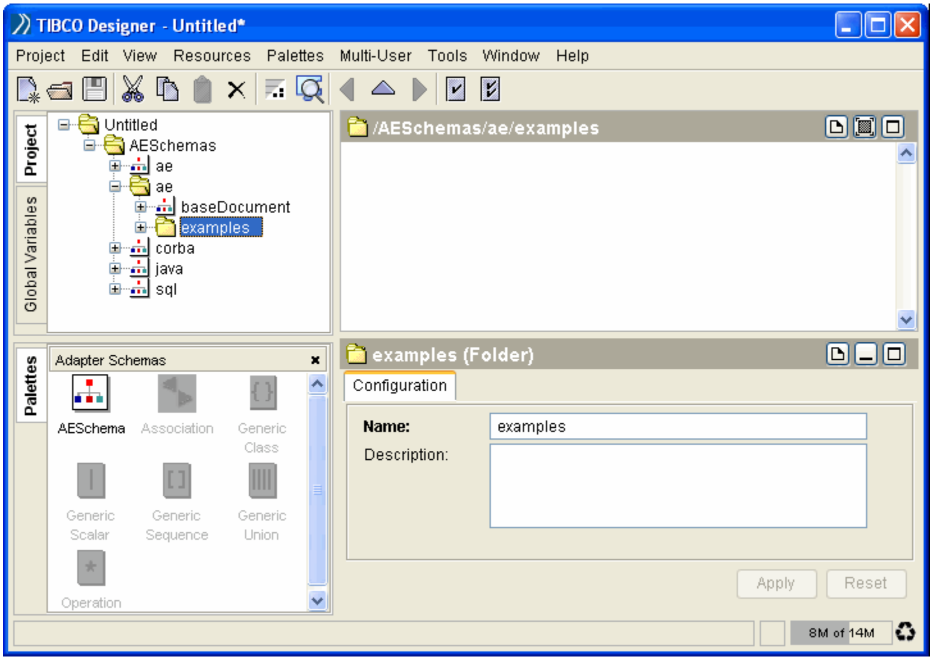
- Task A, Configure Schema, page 178
- Task B, Configure non-Schema Data, page 181
- Task C, Link Schema and Publisher, page 184

Task A Configure Schema

All schema data can be configured using the resources in the `AESchemas` folder. It is recommended to place your schema data for each adapter in a separate folder inside the `AESchemas/ae` folder.

1. Start TIBCO Designer with a new empty project.
2. Go to `AESchemas/ae`, create an `examples` folder.

Figure 26 Create a Folder for Schema Data



TIBCO Designer - Untitled*

/AESchemas/ae/examples

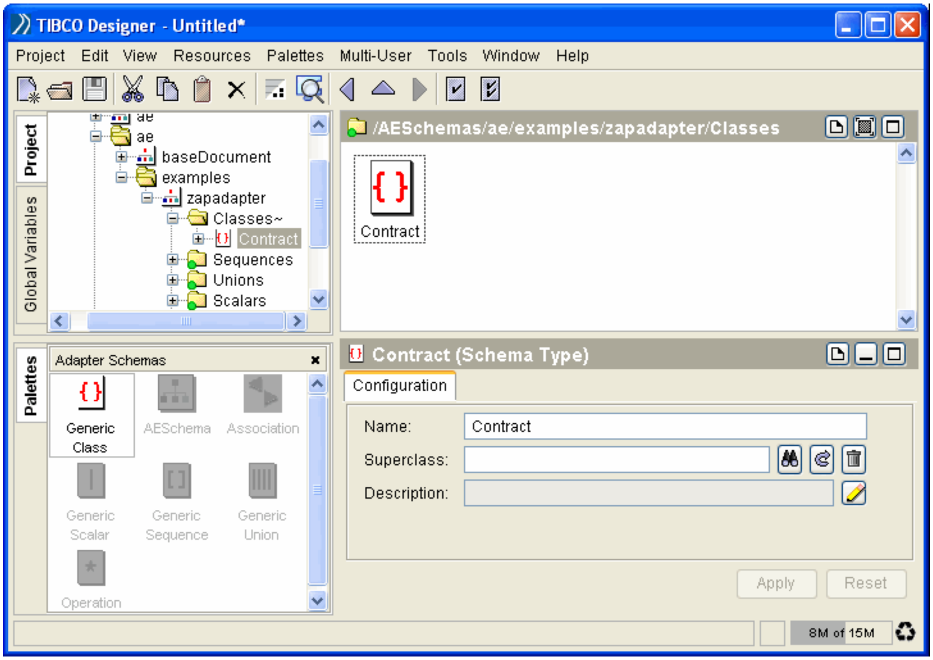
examples (Folder)

Name: examples

Description:

3. Open the new folder and drag an `AESchema` icon from the Palettes panel into the Design panel. Name it `zapadapter` and click **Apply**.
4. Create a new class named `Contract`.
   a. Double-click to select `zapadapter`, then double-click the `Classes` folder.
   b. Drag the `Generic Class` object from the Palettes panel to the Design panel.
   c. In the Class Type filed, select `Schema` and click **Apply**.
   d. Change the name to `Contract`. Click **Apply**.

Figure 27 Create a Class



Name: Contract
Superclass:
Description:

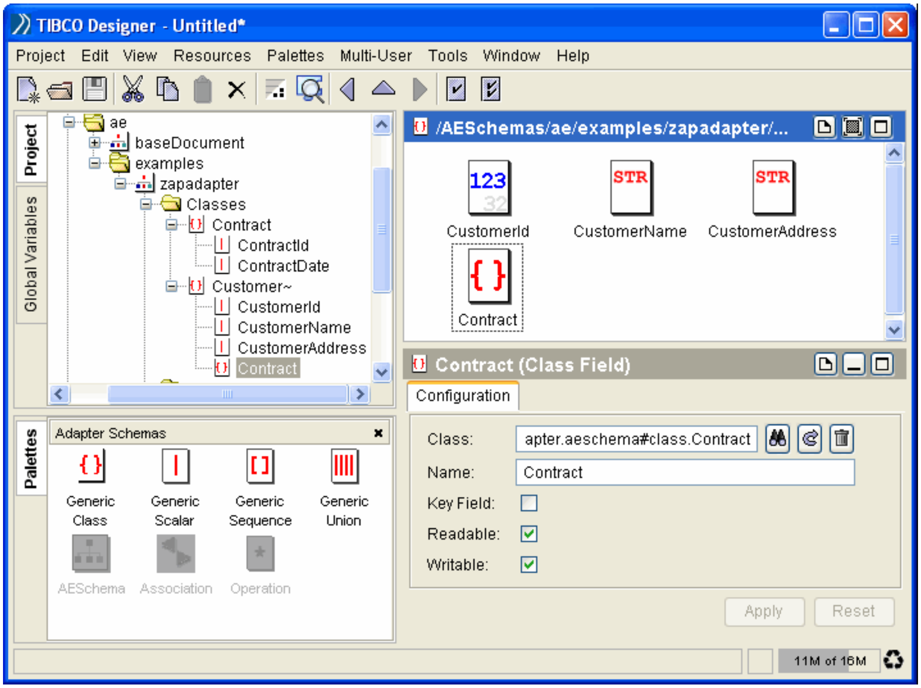
5. Add a `ContractId` attribute.
   a. Double-click the `Contract` class
   b. Drag a `Generic Scalar` object from the Palettes panel to the Design panel.
   c. Change the Name to `ContractId`.
   d. For the Type, choose **i4(32-bit integer)** from the drop-down list.
   e. Click **Apply**.

6. Add a `ContractDate` attribute of type `Date` (same procedure as above).
7. Create another new class named `Customer` with the following attributes:
   - — `CustomerId` (Type i4 32-bit integer)
   - — `CustomerName` (Type string)
   - — `CustomerAddress` (Type string)
8. Add a `Contract` attribute to the class named `Customer`.

   You can drag a `Contract` class from the project tree into the Design panel.

   Alternatively, you can add a generic class attribute and click the Browse button, then select the `Contract` class.

Figure 28 Add an Attribute



Class: apter.aeschema#class.Contract

Name: Contract

Key Field: ☐

Readable: ☑

Writable: ☑

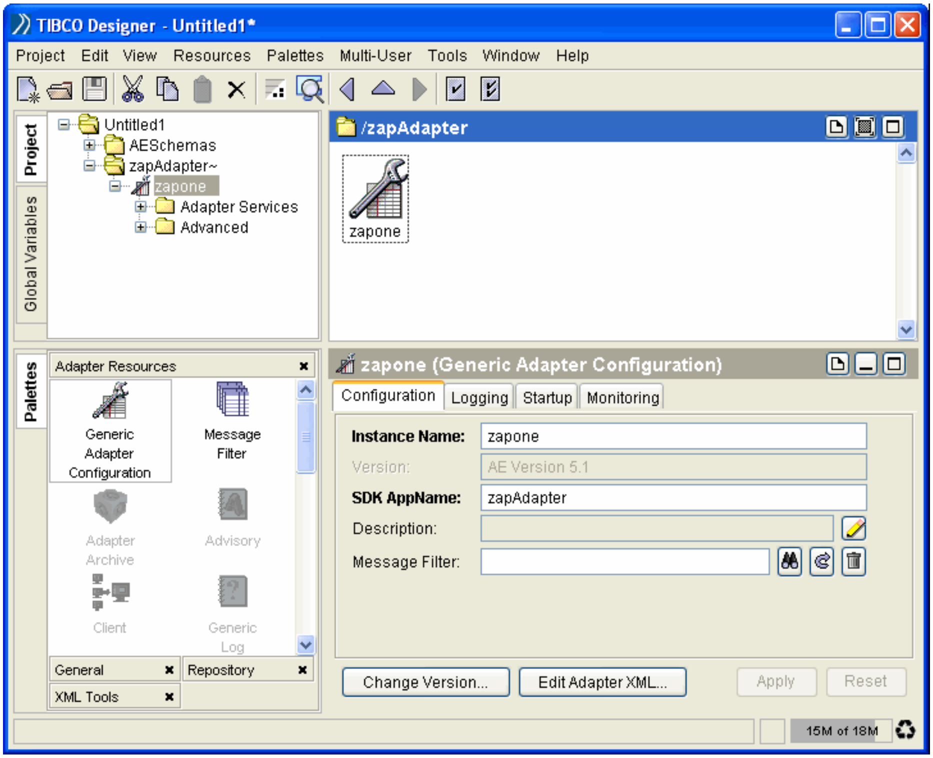
9. Save the configuration.

Task B Configure non-Schema Data

To configure the non-Schema data, that is, the adapter itself and its services, endpoints, and sessions:

1. Drag a folder to the Design palette and name it `zapAdapter`.
2. In the Project panel, double-click the `zapAdapter` folder, then drag a `Generic Adapter Configuration` from the Adapter Resources palette into the Design panel.
3. In the Instance Name field, type **`zapone`**.
4. In the SDK AppName field, type **`zapAdapter`**. This is the short name of the adapter and will be used in the source code. Click **Apply**.

Figure 29 Adapter Configuration



Instance Name: zapone

Version: AE Version 5.1

SDK AppName: zapAdapter

Description:

Message Filter:

5. In the project panel, expand `zapone`, then click the `Adapter Services` folder.
6. Drag a `Publication Service` from the Palettes panel into the Design panel. In the Configuration Name field, change the name to **`zappublisher`**.

7. Select the **Transport** tab. The fields should have the following values:

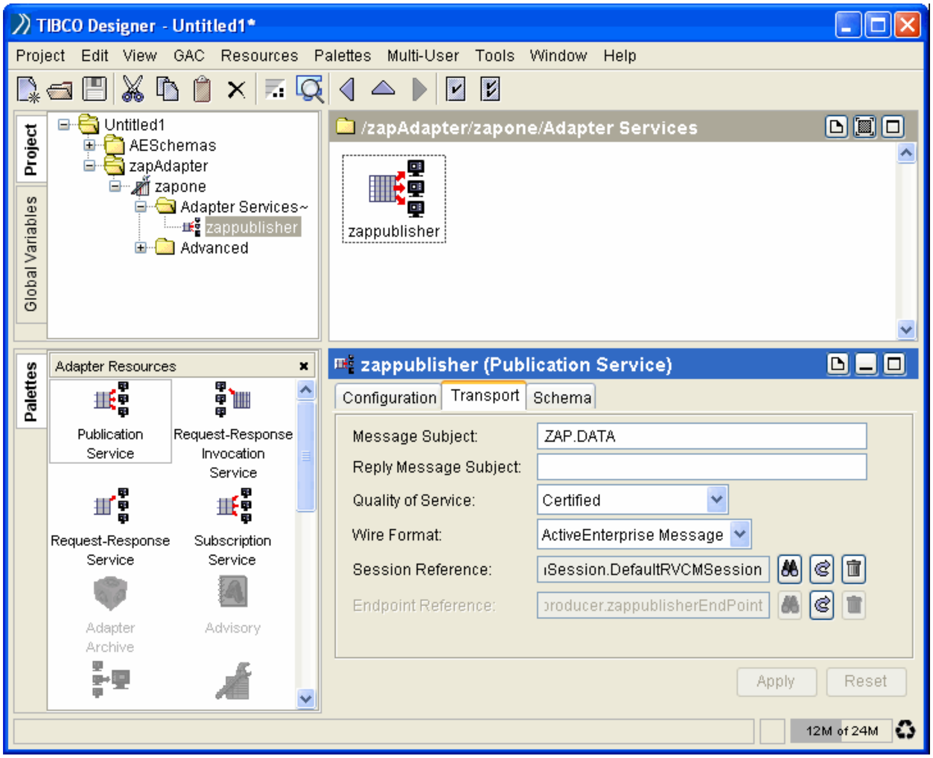
   a. Quality of Service field: `Certified`

   b. Message Subject field, type **`ZAP.DATA`**

   c. Click **Apply**.

   TIBCO Designer creates the service and a corresponding endpoint and session. Endpoint and session are placed in the `Advanced` folder.

Figure 30 Add a Publication Service



Message Subject: ZAP.DATA

Reply Message Subject:

Quality of Service: Certified

Wire Format: ActiveEnterprise Message

Session Reference: Session.DefaultRVCMSession

Endpoint Reference: roducer.zappublisherEndPoint

8. To add a timer, double-click the `Timers` folder (inside the `Advanced` folder) and drag a `Timer` into the Design panel. In the Configuration tab:

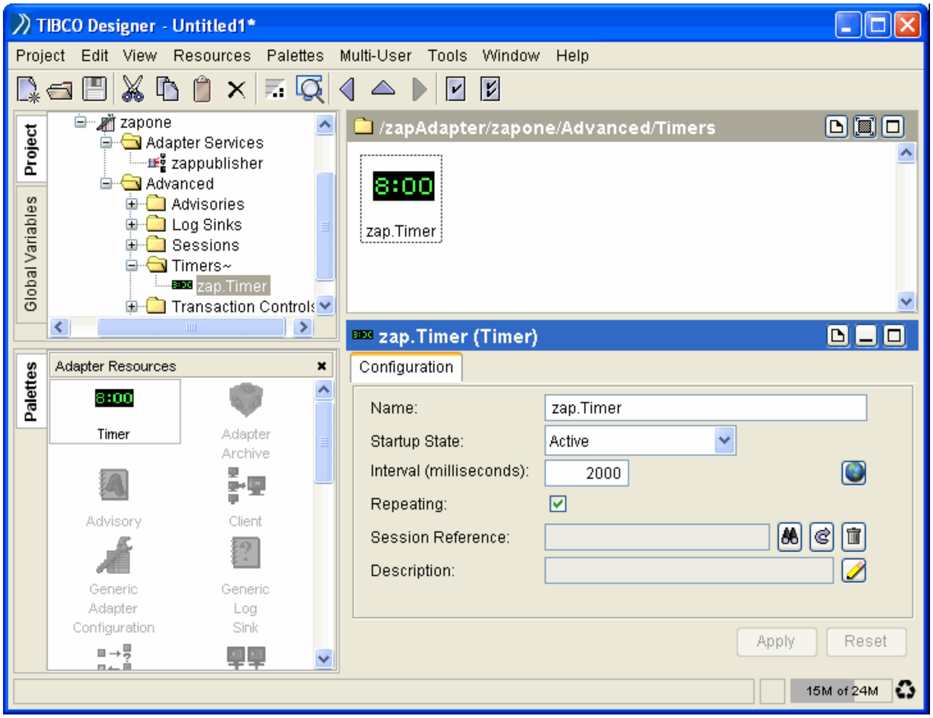
   a. Name field, type **`zap.Timer`**.

   b. Interval field, type `2000` (in millisecond).

   c. Repeating field, leave the check box selected.

   d. Click **Apply**.

Figure 31 Add a Timer



Name: zap.Timer

Startup State: Active

Interval (milliseconds): 2000

Repeating: ☑

Session Reference:

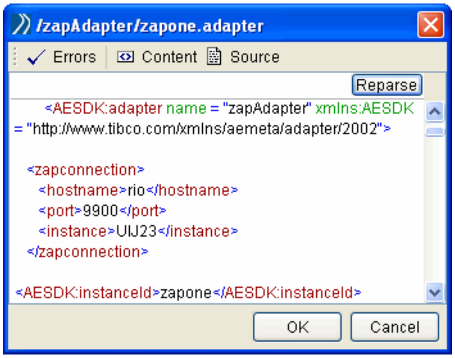
Description:

9. To add custom information, select the **zapone** and click **Edit Adapter XML**.

   Click **Source** and add the following information to the XML file, under `<AESDK:adapter name = "zapAdapter">`:

```
<zapconnection>
   <hostname>rio</hostname>
   <port>9900</port>
   <instance>UIJ23</instance>
</zapconnection>
```

Figure 32 Edit Adapter XML



```
<AESDK:adapter name = "zapAdapter" xmlns:AESDK
= "http://www.tibco.com/xmlns/aemeta/adapter/2002">

<zapconnection>
  <hostname>rio</hostname>
  <port>9900</port>
  <instance>UIJ23</instance>
</zapconnection>

<AESDK:instanceId>zapone</AESDK:instanceId>
```

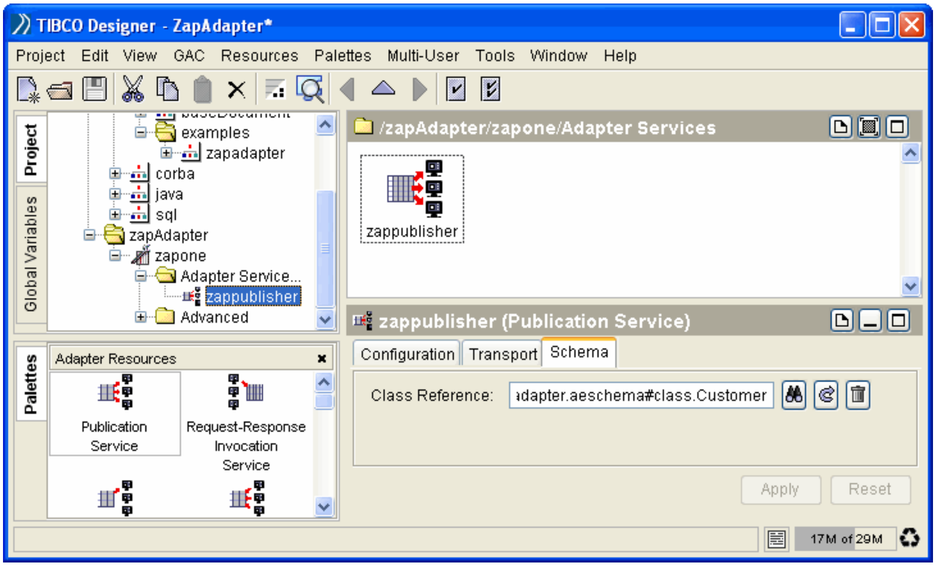
10. Click **OK**.

Task C Link Schema and Publisher

The final step in the adapter configuration is to link the schema to the publisher.

1. In the Project panel, Expand **zapone** > **Adapter Services**.
2. Select `zappublisher` and click the **Schema** tab.
3. Click the **Browse resources** button. In the popup dialog, navigate to the `Customer` class and select it. Click **Apply** and save the project.

Figure 33 Link Schema and Publisher



Class Reference: dapter.aeschema#class.Customer

Implementing the Adapter Code

Configuration information is processed by `MApp` during initialization and used throughout the application. The flow of information is as follows:

1. Specify configuration information using the TIBCO Designer software, which stores the information in the project repository.
2. During execution of the `MApp::start()` method, the SDK stores the configuration information in an `MProperties` object.
3. The application accesses the `MProperties` information as needed.
4. During execution of `MApp::stop()`, the SDK deletes the `MProperties` object. The custom adapter can no longer access the information.

For an example of a `ZapAdapter` application, see the *SDK_HOME*`\Examples` directory.

Chapter 12 Creating a Deployable Custom Adapter

This chapter explains how to create a generic or custom adapter that can be deployed, started, stopped, and monitored using TIBCO Administrator.

Topics

Overview

This chapter shows how to integrate the `Zap Adapter` example with TIBCO Administrator.

The TIBCO ActiveMatrix BusinessWorks process consists of an adapter subscriber listening to the message published by TIBCO Adapter SDK example `zapadapter` and publishing it back using a plain RV publisher.

The process of making this example TIBCO Administrator compliant involves the following steps:

- Modifying Code for TIBCO Administrator Compliance, page 190
- Configuring the Adapter, page 195
- Adding the Adapter to the Domain, page 198

The *SDK_HOME*`\resourceKit` directory includes various programs. Using these programs you can preview some of the latest development in the Java version of TIBCO Adapter SDK. It also demonstrates programs that utilize a *thin* MApp or no MApp. These programs demonstrate special usages of the TIBCO Adapter SDK using Servlet technology.

The programs in *SDK_HOME*`\resourceKit` directory may contain APIs that are not documented and may change in the coming releases.

Source code is provided as is and should be treated as an unsupported program. Comments, feedbacks, and bug reports are welcome but no bug fix release should be expected in any bug that is related to these programs.

Setting Up the Example

Load the provided project into a server-based repository as only a server-based project can be deployed.

1. Start TIBCO Designer.
2. From the Project menu, select **Import Full Project**.
3. In the Local Repository tab, browse to select the project file `zap_compliant.dat` in the directory *SDK_HOME*`\resourceKit\deployableAdapter`. Click OK.
4. Save the project as a server project named `zapadapter`.
5. Compile the example.
   - — For the Java example, compile the Java source code using the `compile.bat` file. The `compile.bat` file generates a `.jar` file and places it under the `../bin` directory. It also copies the `wrap.exe` file to the `../bin` directory and renames it to `zapadapter.exe`.

Make necessary changes to the `CLASSPATH` to successfully compile the example.

   - — For the C++ example, the Developer Studio `.dsw` file generates an executable `zapadapter.exe` and places it under the `../bin` directory.

Modifying Code for TIBCO Administrator Compliance

The following standard TIBCO Hawk microagent methods are required for the adapter to comply with TIBCO Administrator.

- `COM.TIBCO.ADAPTER::getHostInformation()`
- `COM.TIBCO.ADAPTER::getAdapterServiceInformation()`

To use these microagent methods, modify the code as follows:

1. Adding MHostInfo, page 190
2. Adding MAdapterServiceInfo, page 192
3. Implementing Custom Advisory Listener, page 192
4. Implementing Standard ActiveEnterprise Tracing with MMessageBundle, page 193

Adding MHostInfo

The following information is Java-specific. For changes required in the C++ adapter, look at the C++ example code in the *SDK_HOME*`\resourceKit` directory. The main difference is that, for C++ SDK, you need to change code to link with the Service Wrapper library.

The `MHostInfo` contains the application name, instance ID, and state information that is returned to the SDK standard Hawk microagent method `getHostInformation()`. It can also contain backend specific information such as the backend application name, version, connection status, and so on.

1. Create `MHostInfo` in the `ZapAdapter` constructor and set application status (`AppState`) to `INITIALIZING`.

   AppName and Instance ID are automatically set in the constructor from the information provided by the `MApp` parameter.

Application status is not reported to TIBCO Administrator until the standard microagent has been registered with TIBCO Runtime Agent and event dispatch has started.

```
MApp app = new TestAdapterCore( p );
MHostInfo hostInfo = new MHostInfo(app);
  //** set hostInfo service state
  hostInfo.setAppState(MUserApplicationState.INITIALIZING);
  app.setHostInfo(hostInfo);
  app.start();
```

2. Obtain `MHostInfo` back inside `onInitialization()` method and set AppState to RUNNING. After connecting to ZAP, set the backend information obtained. Events dispatch starts after `onInitialization()` is done.

```
onInitialization() {

  //** Create Host information
MHostInfo hostInfo = getHostInfo();
hostInfo.setAppState(MUserApplicationState.RUNNING);
setHostInfo(hostInfo);
    ....

ZAPCustomer.zap_Init();
ZAPConnection conn = new ZAPConnection(this);
if (conn.connect()) {
     //** Trace no tracking id, second param set to null
     getTrace().trace("AEZAP-2000", null,
        "Successfully connected to ZAP database");
     hostInfo.setExtendedInfo("Host", conn.getHostName());
     hostInfo.setExtendedInfo("Port",
         new Integer(conn.getPortNumber()).toString());
setHostInfo(hostInfo);
}
....
}
```

3. Obtain `MHostInfo` back inside `onTermination()` method and set AppState to STOPPED. The `onTermination()` method is called when `MApp.stop()` is called, which happens when the standard Hawk method `stopApplicationInstance()` is invoked or by wrapper shutdown function (shown below).

```
protected void onTermination() throws MException
{
    // update HostInfo to STOPPED
    getTrace().trace("AEZAP-2000", null, "onTermination: stopping
    adapter");
    MHostInfo hostInfo = getHostInfo();
    hostInfo.setAppState(MUserApplicationState.STOPPED);
    setHostInfo(hostInfo);
}
```

4. In the wrapper's shutdown function, (Note that this function is used by `wrapper --stop`), call `MApp.stop()` to shut down the adapter correctly, that is, do the cleanup specified in `MApp.onTermination()`.

```
public void shutdown()
{
try {
  // set service state to stopping
  MHostInfo hostInfo = new MHostInfo(app);
  hostInfo.setAppState(MUserApplicationState.STOPPING);
  app.setHostInfo(hostInfo);
  // Stop adapter, this will call MApp.onTermination()
  app.stop();
  } catch (Exception ex) {
       System.out.println( "shutdown: " + ex);
       ex.printStackTrace();
  }
}
```

Adding MAdapterServiceInfo

Adapter services are defined in terms of a service name, endpoint, and associated schema class(es). Adapters must register all its services either by enumeration of all endpoint components or simply by component name. For each service, the application must call set (serviceName, endpointName, schema) to register the adapter service.

Inside `onInitialization()` retrieves all publisher/subscriber, client/server endpoints with their associated schema and registers them as adapter services using `MAdapterServiceInfo`.

```
onInitialization()
{
...

//** Register publisher service
     MAdapterServiceInfo appInfo = new MAdapterServiceInfo();
     Enumeration classEnum = pub.getClassNames();

     //** Retrieve all class schema specified for this endpoint
     while ( classEnum.hasMoreElements() )
     {
        String schemaname = (String) classEnum.nextElement();
        appInfo.set("Publish service", pub.getName(), schemaname);
     }
setAdapterServiceInfo(appInfo);
...

}
```

Implementing Custom Advisory Listener

Replace the default advisory listener with a custom advisory listener that sends messages to a log file instead of `stdout`.

Depending on the advisory subject, additional action can also be taken. For example, stop the adapter on a RVCM collision advisory. You can then access the log file using the TIBCO Administrator user interface. Note that the user can subclass from MAdvisoryListener and explicitly call the superclass callback directly to extend the default behavior.

To set a custom advisory listener, use the `MApp::setAdvisoryListener()` method.

1. Set the user advisory listener prior to the `MApp.start()` call.

```
...
app = new TestAdapterCore( props );
...
app.setAdvisoryListener(new CustomAdvisoryListener(app));
...
app.start();
```

2. User advisory listener should be implemented in a separate `CustomAdvisoryListener` class. This will log configured advisory messages into log file.

Implementing Standard ActiveEnterprise Tracing with MMessageBundle

To implement standard tracing in the code, the trace and debug methods must use the (errorCode, trackingInfo) form. For Java, the message bundle properties file can be used to store all error codes and messages.

The `zapadapter.properties` file contains the following information:

```
errorRole.Application.AEZAP-0001="Connection to Zap failed with
parameters: %1"
errorRole.Application.AEZAP-0002="Connection to Zap failed or
system unavailable: %1"
errorRole.Application.AEZAP-0003="Request to Zap failed with
parameters: %1"
errorRole.Application.AEZAP-0999="Exception caught: %1"
debugRole.Application.AEZAP-1000="debugTrace: %1"
infoRole.Application.AEZAP-2000="infoTrace: %1"
```

The `zapdapter.properties` file is loaded after MApp is instantiated and before `MApp.start()` is invoked.

```
app = new TestAdapterCore( props );
...
//** load application resource file
MMessageBundle.addResourceBundle("zap", "zapadapter");
```

```
...
app.start();
```

A sample use of the error trace and info trace in `onInitialization()` method is as follows:

```
if (conn.connect())
    {
    //** Trace no tracking id, second param set to null
    getTrace().trace("AEZAP-2000", null, "Successfully connected
    to ZAP database");
    ...
    }
else
    {
    //** Trace with no tracking id
    getTrace().trace("AEZAP-0002", null, "Fail to connect to ZAP
    database");
    ...
    }
```

To include resources that depend on external files, such as the Java Activity in TIBCO ActiveMatrix BusinessWorks, you must create an AliasLibrary. Resources in the project can reference aliases in the AliasLibrary to resolve external file dependencies that they may have at runtime or debug time.

When building an enterprise archive file, the files referenced by the aliases defined in an AliasLibrary that you include in the project are included in the archive file. For more information, refer to the *TIBCO Designer User's Guide*.

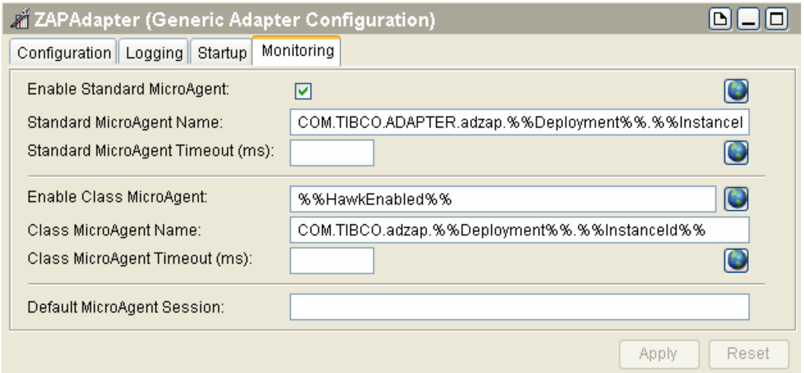
Configuring the Adapter

This section explains how to configure the adapter's monitoring options, log sinks, and advisories in TIBCO Designer.

Monitoring Tab Configuration

1. In the Project panel, expand the folder named `EXAMPLE_ZAPADAPTER`, and then select the adapter configuration named `ZAPAdapter`.
2. Click the **Monitoring** tab. Configure the options in the Monitoring tab as follows.

Figure 34 Monitoring Tab



Enable Standard MicroAgent: ☑
Standard MicroAgent Name: COM.TIBCO.ADAPTER.adzap.%%Deployment%%.%%InstanceI
Standard MicroAgent Timeout (ms):
Enable Class MicroAgent: %%HawkEnabled%%
Class MicroAgent Name: COM.TIBCO.adzap.%%Deployment%%.%%InstanceId%%
Class MicroAgent Timeout (ms):
Default MicroAgent Session:

- **Enable Standard MicroAgent** — Select the check box.
- **Standard MicroAgent Name** — `COM.TIBCO.ADAPTER.`*global_acronym*`.%%Deployment%%.%%InstanceId%%`

  Where *global_acronym* stands for the name that is provided in the domain utility (Global Acronym is set to "adzap") while adding the adapter component to the domain. This name must be globally unique within the TIBCO Administrator domain.
- **Enable Class MicroAgent** — `%%HawkEnabled%%`
- **Class MicroAgent Name** — `COM.TIBCO.`*global_acronym*`.` `%%Deployment%%.%%InstanceId%%`

  Where *global_acronym* stands for the name that is provided in the domain utility (**`adzap`** in this example) while adding the adapter component to the domain. This name must be globally unique within the TIBCO Administrator domain.

— **Default Microagent Session** — If non-default RVD parameters are used for the Hawk session in the Admin server, you must add a separate session in this field. For this session, the TIBHawkDaemon, TIBHawkService, and TIBHawkNetwork RVD parameters must be specified.

To change the session parameter used by the Adapter SDK standard class microagent:

a. Create a RV session with custom parameters for service and daemon port. For example:

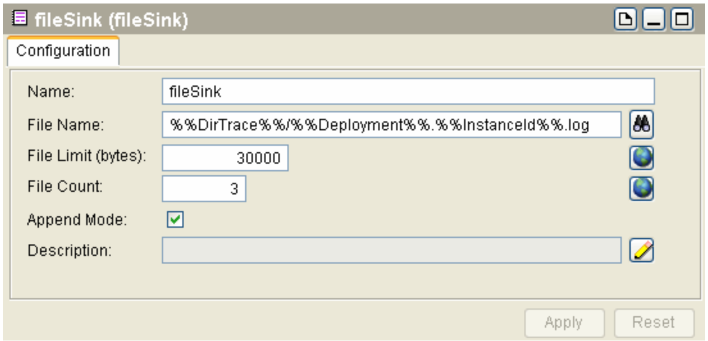
```
<session name="hawkSession" service="9000" daemon="tcp:9000"
/>
```

b. Under the adapter instance, click the **Monitoring** tab and add the name of this newly created session to the Default MicroAgent Session field. Refer to the *TIBCO Designer Resource Management Guide* for details.

Log Sinks Configuration

1. In the Project panel, expand **EXAMPLE_ZAPADAPTER** > **ZAPAdapter** > **Advanced** > **Log Sinks**.
2. Select **fileSink**. In the Configuration tab, ensure that the File Name field is set to `%%DirTrace%%/%%Deployment%%.%%InstanceId%%.log`.

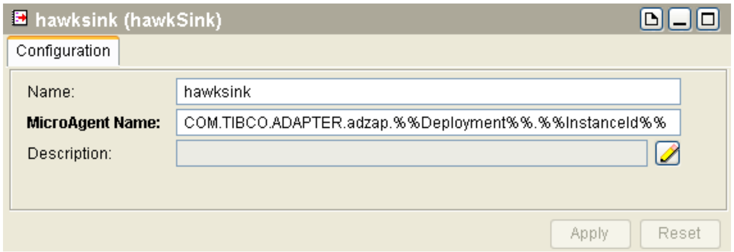
Figure 35 fileSink Configuration



Name: fileSink
File Name: %%DirTrace%%/%%Deployment%%.%%InstanceId%%.log
File Limit (bytes): 30000
File Count: 3
Append Mode: ☑
Description:

3. Select **hawkSink**. In the Configuration tab, ensure that the MicroAgent Name is the same with the MicroAgent name specified in the Monitoring tab.

Figure 36 hawkSink Configuration



Name: hawksink

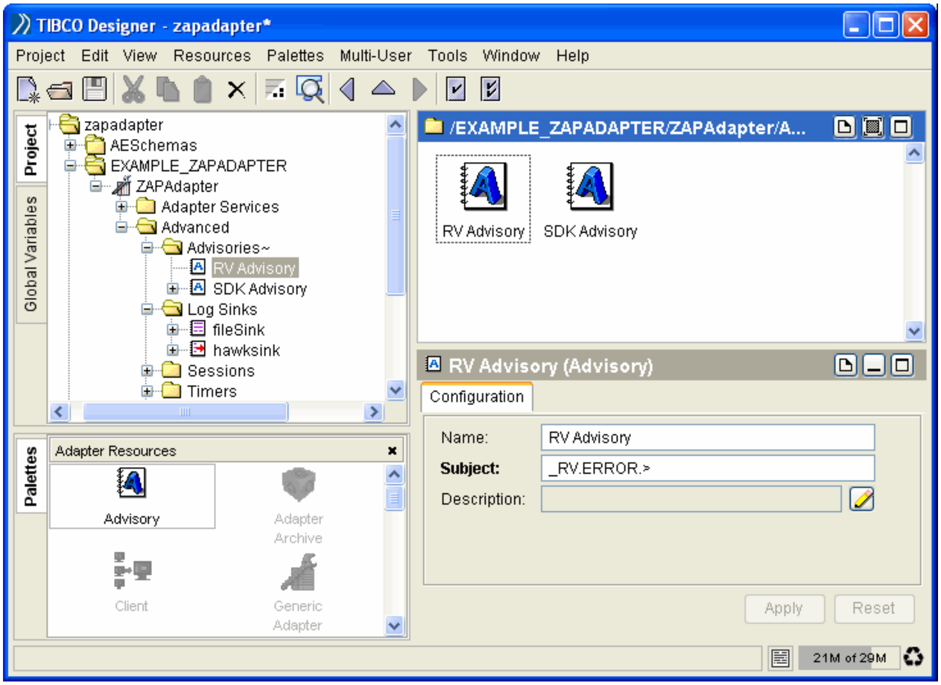
MicroAgent Name: COM.TIBCO.ADAPTER.adzap.%%Deployment%%.%%InstanceId%%

Description:

Advisories Configuration

1. In the Project panel, expand **EXAMPLE_ZAPADAPTER** > **ZAPAdapter** > **Advanced** > **Advisories**.
2. Configure the advisory subject to listen to:
   - — RV Advisory — subject "`_RV.ERROR.>`" and "`_RV.WARN.>`"
   - — SDK Advisory — subject "`_SDK.>`"

Figure 37 Advisories Configuration



Name: RV Advisory

Subject: _RV.ERROR.>

Description:

Adding the Adapter to the Domain

To deploy the custom adapter:

- Creating an Alias Library in TIBCO Designer, page 198
- Creating EAR File in TIBCO Designer, page 198.
- Adding the ZapAdapter to the TIBCO Administrator Domain, page 200.
- Creating the ZapAdapter Application in the TIBCO Administrator Domain, page 201.
- Deploying, Starting, and Stopping the Adapter, page 203.

Creating an Alias Library in TIBCO Designer

You can include the external files in the EAR by creating an Alias Library. For detailed steps, refer to *TIBCO Designer's User's Guide*.

To bundle an Alias Library:

1. Start TIBCO Designer.
2. From the Edit menu, select **Preferences**.
3. In the TIBCO Designer Preferences dialog, click the **File Aliases** tab and create aliases for the external files.
4. Add an AliasLibrary and include these external files in the **Aliases** tab, check **Deploy?**, and click **Apply**.
5. Select **Shared Archive** and add this AliasLibrary to the Resources tab.

Creating EAR File in TIBCO Designer

To deploy a project, you must generate an Enterprise Archive for it. Before creating an Enterprise Archive (EAR) file, in Designer, you must convert the existing AE `.dat` file to multi-file format, and open the multi-file project.

Task A Convert the Existing File

To convert the existing example `.dat` file:

1. Open TIBCO Designer. In the TIBCO Designer startup window, click the **Administration** tab.
2. Click **Convert DAT to Files**.

3. In the Convert dialog:
    a. In the DAT File field, click **Browse** and navigate to *SDK_HOME*`\resourcekit\deployableAdapter\zap_compliant.dat`.
    b. In the Project Directory field, type *install-path*`\tibco\tra\`*version_num*`\resourcekit\sdk\deployableAdapter\zapadapter_multifiles`.
    c. Click **OK**. TIBCO Designer converts the `.dat` file to a multi-file project.

The name of the `.dat` file cannot be used as the name of the directory.

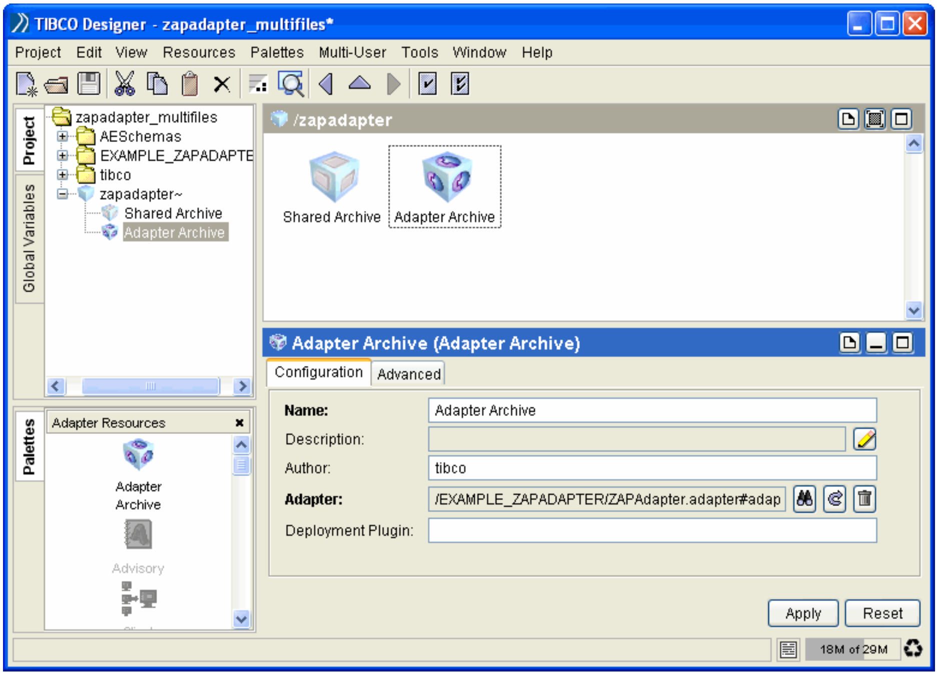
4. Click the **Project** tab and choose **Open Existing Project**.
5. In the Open Project dialog, click the **Multi-File Project** tab, browse to select the project directory *SDK_HOME*`\resourcekit\deployableAdapter\zapadapter_multifiles`.
6. Click **OK**.

Task B Build an Enterprise Archive

To build the archive:

1. In the Project panel, select the top-level (project) resource.
2. In the Palettes panel, click the **General** palette, and then drag an **Enterprise Archive** into the Design panel.
3. In the Configuration panel, replace the default value in the Name field with `zapadapter` and click **Apply**.
4. In the Project panel, expand **zapadapter**, select **Shared Archive**.
5. Click the **Resources** tab. Browse and add the **AliasLibrary** that was created earlier as explained in Creating an Alias Library in TIBCO Designer on page 198.
6. Add an Adapter Archive file:
    a. In the Project panel, select the `zapadapter` enterprise archive. In the Palettes panel, select an Adapter Archive (Adapter Resources palette) and drag it into the Design panel.
    b. In the Configuration panel, click the **Browse Resources** icon next to the **Adapter** field, select the resource from the pop-up, and click **Apply**.

Figure 38 Add an Adapter Archive



Name: Adapter Archive

Description:

Author: tibco

Adapter: /EXAMPLE_ZAPADAPTER/ZAPAdapter.adapter#adap

Deployment Plugin:

c. Click the **Advanced** tab, specify Software Type as `adzap`. Click **Apply**.

7. Save the project.
8. In the Project panel, select the `zapadapter` enterprise archive. Enter the following in the File Location field:
   SDK_HOME`\resourcekit\deployableAdapter\zapadapter.ear`.
9. Click **Build Archive**.

Now, the archive file is ready for deployment.

Adding the ZapAdapter to the TIBCO Administrator Domain

Custom software such as adapters built using the TIBCO Adapter SDK, must be added manually before you can deploy the application.

To add the ZapAdapter software to a TIBCO Administrator domain:

1. Start the TIBCO Administrator user interface and log into the administrator domain in which you want to deploy the application.
2. Select the **Installed Software** console of TIBCO Administrator.
3. Click **Add Custom Software**.

4. Select the machine on which you want to add the custom software and click **OK**.
5. In the panel that is displayed, provide the following information and click **OK**.

Table 40 Add Custom Software

Field	Description
Machine	Name of the machine on which the software is to be added. Click **Change** to add the software on a different machine.
Software Type	Must match software type used to build the EAR file. Enter **`adzap`**.
Software Display Name	The name that should be displayed in TIBCO Administrator. Enter **`ZAP Adapter`**.
Version	This number must match the EAR files loaded for the software. If the EAR file specifies a later version, it cannot be loaded. Enter **`5.6.0`**.
Executable (Full Path)	Executable for this custom software.
Software is an adapter	Select this check box if the custom software is an adapter; clear otherwise.
Java Software	Select this check box for a Java deployable adapter.
Java Start Class	Provide the Java start class. Enter **`ZapAdapter`**.
Java Start Method	Provide the Java start method. Enter **`main`**.
Java Stop Method	Provide the Java stop method. Enter **`shutdown`**.
Java Classpath	Provide the Java classpath.

6. Click **Save**.

Creating the ZapAdapter Application in the TIBCO Administrator Domain

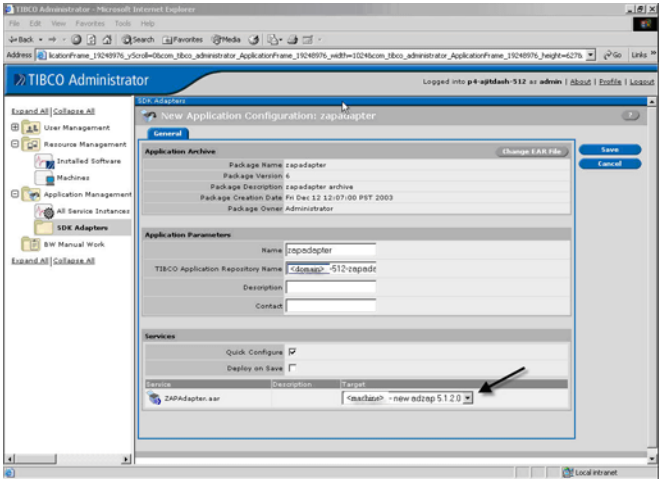
1. Start the TIBCO Administrator user interface and log into the administrator domain in which you want to deploy the application.
2. Click the **Application Management** module and click **New Folder**. In the window that displays, type **`SDK Adapters`** in the Name field. Click **Save**.

It is recommended to create folders for complex applications. Using folders is not required.

3. In the left-hand panel, click the SDK Adapter folder, click **New Application**.
4. Click **Browse**, select the `zapadater Enterprise Archive` file created in the first step.

   Make sure that an adapter is available as the Target in the bottom right corner. If it isn't, register it first. See Adding the ZapAdapter to the TIBCO Administrator Domain on page 200 for more information.

Figure 39 Create New Application



New Application Configuration: zapadapter

Application Archive

Package Name: zapadapter
Package Version: 6
Package Description: zapadapter archive
Package Creation Date: Fri Dec 12 12:07:00 PST 2003
Package Owner: Administrator

Application Parameters

Name: zapadapter
TIBCO Application Repository Name: <domain> -512-zapadc
Description:
Contact:

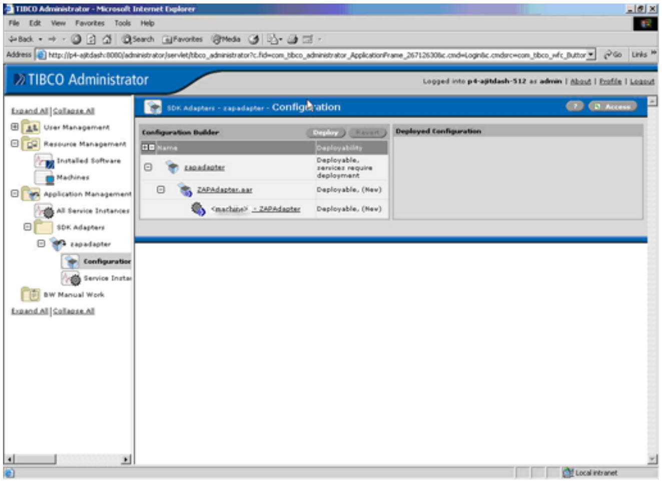
Services

Quick Configure ☑
Deploy on Save ☐

Service	Description	Target
ZAPAdapter.aar		<machine> - new adzap 5.1.2.0

5. The next dialog displays the Configuration console with Configuration Builder on the left and the Deployed Configuration pane on the right. The Deployed Configuration pane is empty because the application has not yet been deployed.

Figure 40 Configuration Console


TIBCO Administrator - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://p4-ajtdash:8080/administrator/servlet/tibco_administrator?c.fid=com_tibco_administrator_ApplicationFrame_267126308c.cmd=Login&c.cmdsrc=com_tibco_wfc_Button
Go Links
TIBCO Administrator
Logged into p4-ajtdash-512 as admin | About | Profile | Logout
Expand All | Collapse All
User Management
Resource Management
Installed Software
Machines
Application Management
All Service Instances
SDK Adapters
zapadapter
Configuration
Service Insta
BW Manual Work
Expand All | Collapse All
SDK Adapters - zapadapter - Configuration
?
Access
Configuration Builder
Deploy
Revert
Deployed Configuration
Local intranet

Name	Deployability
zapadapter	Deployable, services require deployment
ZAPAdapter.aar	Deployable, (New)
<machine> - ZAPAdapter	Deployable, (New)

6. Click **Deploy**.

For instructions on how to deploy an adapter, see the *TIBCO Administrator's Guide*.

Deploying, Starting, and Stopping the Adapter

Once the adapter has been added to the domain and deployed, it can be started or stopped using the TIBCO Administrator.

Administrator calls the class Hawk microagent to stop a deployed adapter that is independent of the `stop()` method, which is called by the wrapper when it was run as an NT service. On UNIX, there is no NT service, so the `stop()` method that is used to update NT service status is irrelevant. In other words, a wrapper-enabled and deployed adapter has two ways of running and terminating on Windows:

- Launched and terminated as an NT service and calls the `stop()` method to `shutdown()`.
- Launched and terminated by Administrator, in this case, the shutdown function is not called unless explicitly coded in `onTermination()`.

For additional information, see the *TIBCO Administrator's Guide*.

Running the Example

If the domain registration and deployment are successful, the adapter can be launched from TIBCO Administrator.

In the Designer window, run the `zapsubscriber` process to receive any messages published.

In a separate command window, launch `tibrvlisten zap.out`.

Expected Results

You should see the output in the `tibrvlisten` window.

Chapter 13 TIBCO Wrapper Utility

This chapter details the steps required to configure and run adapters using the TIBCO Wrapper utility, which allows Unix and Microsoft Windows operating systems to restart an adapter when the system reboots. For C++ applications, it also explains the build and coding requirements.

Topics

Overview

The TIBCO Wrapper utility allows deployment of an adapter as a service. Among other benefits, the wrapper allows the operating system to start the adapter automatically upon reboot, regardless of the operating system in use.

- On Microsoft Windows platforms, a *wrapped* application can be installed and run as a Windows Service.
- On UNIX platforms, the functionality serves as a process abstraction. Hooks for TIBCO Administrator mean that *wrapped* applications can be restarted on reboot under UNIX and monitored using TIBCO Administrator.

The TIBCO Wrapper utility can be used for both Java or C++ adapters. The tool uses standard Java properties files for configuration of Service and Java Virtual Machine (JVM) parameters.

When you invoke the Wrapper, it looks in the current directory for the *appname*`.tra` file. If that file is not found, it searches the PATH for *appname*`.tra`.

If a `.tra` file is not found in one of those locations, the Wrapper cannot start.

Running an Adapter as a Microsoft Windows Service

This section explains how to run an adapter as a Microsoft Windows service.

Java Adapters

To run a Java adapter as a service under Microsoft Windows:

1. Ensure that the adapter code is updating the application state information (see `MHostInfo` in the online API documentation).
2. Rename the `wrap.exe` or `gwrap.exe` binary to the name of your application (for example, `SimpleApp.exe`).
3. Create a properties file detailing Service and Java properties. See Wrapper Properties on page 214.

    The properties file should use standard Java properties file syntax.

    Name the properties file *app_name*`.tra` (for example, `SimpleApp.tra`). It will be picked up automatically when one of the TIBCO Wrapper command-line options is used. You can also specify which properties file to use through the `--propFile` *filename* command-line option.

4. Install the service with the Service Control Manager by using the `--install` command line option.

    ```
    SimpleApp.exe --install
    ```

5. Start the service using one of these options:

    — through the Control Panel

    — using the `--start` command line option

    Upon reboot, you don't have to restart automatic services explicitly.

    The application is now running in the background as a service.

If you changed details in the properties file, run `--uninstall` and `--install` again. Otherwise, changes do not take effect.

Note that the properties file itself is *not* read when the application is running as a Windows Service.

C++ Adapters

Use the TIBCO Wrapper with the C++ application in two ways:

- Explicitly Linking the Library—This produces an executable that can be run from the command line.
- Delayed Application Shared Library Loading—At times, it is necessary to set a per-application PATH instead of accepting a system setting. The PATH environment variable dictates which dependent shared libraries are loaded.

  This approach produces a shared library that can then be loaded by the Wrapper executable, and allows you to do this.

Explicitly Linking the Library

To run a C++ adapter as a service on Microsoft Windows platforms:

1. Rename the current `main()` to `AppMain()`.
2. Write an `AppStop()` method, if required.
3. Write a new `main()` that calls the `LaunchWrapper()` function.
4. Modify the existing adapter code to call the `SetServiceState()` function as appropriate for the adapter.
5. Compile and link with the `libwrap.lib` static library.
6. Create a properties file detailing Service properties.

> The properties file should use standard Java properties file syntax (even for C++ custom adapters).
>
> If you named the properties file *app_name*`.tra` (for example, `SimpleApp.tra`), it is picked up automatically. You can also specify which properties file to use through a command-line option.

7. Install the service with the Service Control Manager by using the `--install` command line option.

   ```
   SimpleApp.exe --install
   ```

8. Start the service using one of these options:

   — through the Control Panel applet

   — using the `--start` command line option

   Upon reboot, automatic services are restarted.

The application is now running in the background as a service.

If you changed details in the properties file, run `--uninstall` and `--install` again. Otherwise, changes do not take effect.

Note that the properties file itself is *not* read when the application is running as a Windows service.

Delayed Application Shared Library Loading

To run a C++ adapter as a service on Microsoft Windows platforms, and change the loading of dependent shared libraries to allow the wrapper to modify the PATH environment variable:

1. Change the makefile to produce a shared library target instead of an executable.
2. Rename the current `main()` to `AppMain()`.
3. Write an `AppStop()` method, if required.
4. Write a `RegisterWrapperEntryPoints()` function. This function gives you a function callback pointer for setting the service state.
5. Write a `SetServiceState()` function, as appropriate for the adapter, that uses the pointer obtained from `RegisterWrapperEntryPoints()`.
6. Create a properties file detailing Service properties. Be sure to set the `application.library` property to point to the application shared library.

   The properties file should use standard Java properties file syntax (even for C++ custom adapters).

   If you named the properties file *app_name*`.tra` (for example, `SimpleApp.tra`), it is picked up automatically. You can also specify which properties file to use through a command-line option.

7. Rename the `wrap.exe` or `gwrap.exe` binary to the name of the application (for example, `SimpleApp.exe`).
8. Install the service with the Service Control Manager by using the `--install` command line option.

   ```
   SimpleApp.exe --install
   ```

9. Start the service using one of these options:

   — through the Control Panel applet

   — using the `--start` command line option

   Upon reboot, automatic services are restarted.

The application is now running in the background as a service.

If you changed details in the properties file, run `--uninstall` and `--install` again. Otherwise, changes do not take effect.

Note that the properties file itself is *not* read when the application is running as a Windows service.

Using the TIBCO Wrapper Under UNIX

Some of the facilities available under Microsoft Windows are not available under UNIX because there is no Service Control Manager. However, you have the following benefits:

- Monitoring and management through TIBCO Administrator (requires code modification, see Source Code Changes on page 212).
- Use of a properties file that includes any of the wrapper-settable properties that do not start with `ntservice`.
- An application using the TIBCO Wrapper that has joined the TIBCO administration domain will autostart upon reboot by the TIBCO Administration Server.
- Use of the following command-line options:

```
--propFile
--propVar
--run
--help
--version
--pid
```

Under HP-UX, the wrapper executable and library are explicitly linked against `libjvm.sl`. Because that library uses thread local storage (TLS), it cannot be loaded dynamically through `dlopen`. (C++ adapters also need to link the `libjvm` if they use `libwrap`.) For this reason, the `SHLIB_PATH` environment variable must be set before launching the wrapper.

See the `dlopen` man page on HP-UX for details on the restriction.

Source Code Changes

To use the TIBCO Wrapper, ensure that you link in the wrapper libraries under C++. In addition, you need to make the following source code changes.

Java Only

Adapters need only be certain that they are updating the application's state by using the `MHostInfo` class. This is how the Win32 Service Control Manager (SCM) get notified that the application is running. State change notifications are required, otherwise the SCM will regard the service as non-functional. This may lead to unexpected behavior.

For example:

```
main () {
//** 0) create MApp
   app = new MyMApp(appProperties);
//** 1) create MHostInfo
   MHostInfo hostInfo = new MHostInfo(app);
//** 2) set hostInfo service state
   hostInfo.setAppState(MUserApplicationState.INITIALIZING);
//** 3) associating MHostInfo to MApp
   app.setHostInfo(hostInfo);
   app.start();
}
onInitialization {
    MHostInfo hostinfo = MyMApp.getHostInfo();
    hostInfo.setAppState (MUserApplicationState.RUNNING);

}
```

You should create `MHostInfo` the same place you create `MAppProperties` so that you can set the "Initializing" state before setting "Running" state inside `onInitialization()`.

Make sure that the application state is set for failure and exit conditions. Before the adapter exits, the state should be set first to `STOPPING` and then to `STOPPED`. While there is some facility in the TIBCO Wrapper tool to detect or correct inconsistent state changes, the application itself is responsible for setting state back to the SCM correctly.

Note that the tool can be used in conjunction with existing Java adapters without modification, provided they are updating the application state as noted above. Otherwise, this code needs to be added before attempting to install the adapter as a Service.

C++ Only

Adapters need to rename their current `main()` to `ApplicationMain()` and call the `LaunchWrapper()` method while passing their `ApplicationMain()` method to the wrapper.

In addition, the service state for C++ applications must be set using the wrapper function `SetServiceState()` instead of relying on `MApp::setHostInfo()`.

Wrapper Sample Code

This section shows a sample C++ application code fragment that integrates the wrapper.

```
void AppMain( int argc, char * argv )
{
   SetServiceState( SVC_INITIALIZING );
   // your adapter startup code goes here
   // NOTE: typical adapters will set the service state
   // to RUNNING inside of MApp::onInitialization()
   // instead of here. An exception would be adapters
   // that pass Mfalse to MApp::start().
   //…
   SetServiceState( SVC_RUNNING );
//…
}

void AppStop()
{
   SetServiceState( SVC_STOPPING );
   // your app shutdown code here…
   // Note that it may be more appropriate to set state
   // in MApp::onTermination().
   SetServiceState( SVC_STOPPED );
}

void main( int argc, char * argv )
{
   LaunchWrapper( argc, argv, &AppMain, &AppStop );
}
```

Wrapper Properties

Table 41 lists the predefined properties you can use in the properties file. You can also add application-defined properties to this file. See Two Types of Properties on page 40.

Wrapper Settable Properties

All properties starting with `ntservice` are available only under Microsoft Windows. All other properties are platform independent.

Table 41 Wrapper Settable Properties

Property Name	Presence	Value	Notes
`ntservice.name`	Required on MS Windows	any	Name of this service in the SCM service database.
`ntservice.displayname`	Required on MS Windows	any	Name displayed to users of the Control Panel Services Applet for this service.
`ntservice.binary.path.absolute`	Required on MS Windows	any	Absolute path to the wrapper binary itself. Required when started as a service. Installing the service and then moving the binary is not recommended. Note that the back slash path separator must be escaped with addition back slash (or you can simply use the forward slash). See Examples for ntservice.binary.path.absolute on page 219.
`ntservice.dependencies`	Optional	any	A comma-separated list of dependent services that must be started before starting this service.
`ntservice.starttype`	Optional	`automatic`, `manual`, or `disabled`	Defaults to `automatic` if not set.

Table 41 Wrapper Settable Properties (Cont'd)

Property Name	Presence	Value	Notes
`ntservice.interactive`	Optional	`true` or `false`	Defaults to `true` if not set. Must be `false` to use a specific account and password. Specifying `true` and specifying an account and password is considered illegal by the SCM.
`ntservice.account`	Optional	any	Be certain that the network domain is specified as part of the account. Otherwise, the SCM will reject the account as invalid when attempting to install the service.
`ntservice.password`	Optional	any	Password for the specified account. You can use a password that has been encrypted. In that case, prefix it with #! (pound exclamation mark).
`application.start.dir`	Optional	any	Used to specify a different directory than the current directory to launch the application. Useful when applications make use of relative paths.
`application.args`	Optional	any	Arguments supplied to the application when it launches. The string is exactly as it would be typed on the command line, however, you must escape certain characters that terminate a Java property. Refer to standard Java property file syntax documentation for characters requiring escaping. To read a properties file that specifies SDK properties, include `application.args=-system:propFile <SDK_Properties_file> ....`
`java.start.class`	Optional		The TIBCO Wrapper looks for the `start()` method on this class, which can also be specified directly via the `java.start.method` property. If the `start()` method is not supplied, then the standard Java static `main()` method is used.

Table 41 Wrapper Settable Properties (Cont'd)

Property Name	Presence	Value	Notes
`java.start.method`	Optional		The TIBCO Wrapper attempts to invoke this static method on the start class. If the `start()` method is not supplied, then the standard Java static `main()` method is used.
`java.stop.method`	Optional		If this property is supplied, the utility will attempt to invoke this static method on the start class upon shutdown.
`java.class.path`	Optional	any standard classpath	Standard java classpath that will be passed along to the JVM without modification. Note that the TIBCO Wrapper only uses either `tibco.class.path.extended` or this property , but not both. If both are present, the utility takes `tibco.class.path.extended` instead of this property.
`tibco.class.path.extended`	Optional	any	Classpath that is expanded inline by the utility. Any directory is searched for `.class`, `.jar`, or `.zip` files and expands the final classpath to include all of these files for that directory. Any specific files in the classpath are passed along unaltered.
`java.library`	REQ		Absolute path to the JVM to use (see Sample Properties File on page 220).
`java.heap.size.max`	Optional	4M, 32M, etc.	The syntax is the same as for the extended property (`-Xmx`). For example, "32M".
`java.thread.stack.size`	Optional	4M, 32M, etc.	The syntax is the same as for the extended property (`-Xss`).
`java.property.<prop>`	Optional	any	Allows setting of any Java properties to be passed along to the Java application. Takes the form of `Java.property.foo=bar`. There may be an unlimited number of these per properties file, within file system limitations.

Table 41 Wrapper Settable Properties (Cont'd)

Property Name	Presence	Value	Notes
`ntservice.interactive`	Optional	`true` or `false`	Defaults to `true` if not set. Must be `false` to use a specific account and password. Specifying `true` and specifying an account and password is considered illegal by the SCM.
`ntservice.account`	Optional	any	Be certain that the network domain is specified as part of the account. Otherwise, the SCM will reject the account as invalid when attempting to install the service.
`ntservice.password`	Optional	any	Password for the specified account. You can use a password that has been encrypted. In that case, prefix it with #! (pound exclamation mark).
`application.start.dir`	Optional	any	Used to specify a different directory than the current directory to launch the application. Useful when applications make use of relative paths.
`application.args`	Optional	any	Arguments supplied to the application when it launches. The string is exactly as it would be typed on the command line, however, you must escape certain characters that terminate a Java property. Refer to standard Java property file syntax documentation for characters requiring escaping. To read a properties file that specifies SDK properties, include `application.args=-system:propFile <SDK_Properties_file> ....`
`java.start.class`	Optional		The TIBCO Wrapper looks for the `start()` method on this class, which can also be specified directly via the `java.start.method` property. If the `start()` method is not supplied, then the standard Java static `main()` method is used.

Table 41 Wrapper Settable Properties (Cont'd)

Property Name	Presence	Value	Notes
`java.start.method`	Optional		The TIBCO Wrapper attempts to invoke this static method on the start class. If the `start()` method is not supplied, then the standard Java static `main()` method is used.
`java.stop.method`	Optional		If this property is supplied, the utility will attempt to invoke this static method on the start class upon shutdown.
`java.class.path`	Optional	any standard classpath	Standard java classpath that will be passed along to the JVM without modification. Note that the TIBCO Wrapper only uses either `tibco.class.path.extended` or this property , but not both. If both are present, the utility takes `tibco.class.path.extended` instead of this property.
`tibco.class.path.extended`	Optional	any	Classpath that is expanded inline by the utility. Any directory is searched for `.class`, `.jar`, or `.zip` files and expands the final classpath to include all of these files for that directory. Any specific files in the classpath are passed along unaltered.
`java.library`	REQ		Absolute path to the JVM to use (see Sample Properties File on page 220).
`java.heap.size.max`	Optional	4M, 32M, etc.	The syntax is the same as for the extended property (`-Xmx`). For example, "32M".
`java.thread.stack.size`	Optional	4M, 32M, etc.	The syntax is the same as for the extended property (`-Xss`).
`java.property.<prop>`	Optional	any	Allows setting of any Java properties to be passed along to the Java application. Takes the form of `Java.property.foo=bar`. There may be an unlimited number of these per properties file, within file system limitations.

Table 41 Wrapper Settable Properties (Cont'd)

Property Name	Presence	Value	Notes
`java.extended.properties`	Optional	any	Allows setting of arbitrary extended JVM options (for example, `-XServer -Xdebug`).
tibco.env.<varname>	Optional	any env. variable value	Overrides the environment variable is set. Otherwise, creates it and sets it for the application. Can be used to define any environment variable. For example defining `tibco.env.PATH` will override the existing PATH environment setting. See tibco.env Note below.

tibco.env Note

The `tibco.env.<varname>` property allows you to override the PATH environment variable for C++ custom adapters running as a Windows Service.

To support this, link the shared libraries that you want to be loaded through the overridden PATH with the MSVC /DELAYLOAD linker option. This allows the overridden PATH to take effect before the operating system attempts to load the dependent shared libraries.

- Example for Windows

```
tibco.env.PATH
c:/tibco/tra/5.6/bin;c:/tibco/tpcl/5.6/bin;c:/tibco/tibrv/8.1/bin;
c:/tibco/adapter/zap/bin
```

or

```
tibco.env.PATH
c:\\tibco\\tra\\5.6\\bin;c:\\tibco\\tpcl\\5.6\\bin;c:\\tibco\\tibr
v\\8.1\\bin;c:\\tibco\\adapter\\zap\\bin
```

- Example for UNIX

```
tibco.env.PATH
/opt/tibco/tra/5.6/bin:/opt/tibco/tpcl/5.6/bin:/opt/tibco/tibrv/8.
1/bin:/opt/tibco/zapadapter/bin
```

Examples for ntservice.binary.path.absolute

```
ntservice.binary.path.absolute
c:\\tibco\\adapter\\zap\\bin\\zapadapter.exe
```

or

```
ntservice.binary.path.absolute c:/tibco/adapter/zap/bin/zapadapter
```

Properties Files

Adapters that use the TIBCO wrapper can use a separate SDK properties file. To do so, they specify the SDK properties file in the `application.args` property of the TIBCO Wrapper properties file as follows:

```
application.args=-system:propFile <SDK_Properties_file> ...
```

You can use separate properties files for SDK properties and TIBCO Wrapper properties for clarity. You can also place all properties into the wrapper properties file.

Sample Properties File

This section lists a sample properties file for Microsoft Windows.

```
# ***** some java application properties *****
java.start.class SimpleApp
java.start.method=main

# note that this supports inline directory classpath expansion
java.class.path=.;c:\projects\maverick-dev-rv6\examples\cpp\wrap

#java.library=C:\tibco\JRE\1.5.0\bin\hotspot\jvm.dll
java.heap.size.max 32M

application.start.dir
c:\projects\maverick-dev-rv6\examples\cpp\wrap

# ***** properties for installing as a service *****
ntservice.name=AAA_wrap
ntservice.displayname=AAA Simple Java Service
ntservice.starttype=manual
ntservice.binary.path.absolute=c:\projects\maverick-dev-rv6\exampl
es\cpp\wrap\release\wrap.exe
#ntservice.dependencies=foo bar

ntservice.interactive=true
# NOTE - if interactive is true, then the system account must be
used...
#ntservice.account=
#ntservice.password=
```

Command Line Options

Table 42 and Table 43 list command-line options you can use in conjunction with the TIBCO Wrapper.

Table 42 Command-line Options (all platforms)

Option	Description
`--propFile <properties file>`	By default, the wrapper looks for a properties file with the same name as the binary, with a `.tra` extension. For example, `MyApplication.exe` would look for a properties file name `MyApplication.tra`. This command line option overrides this behavior and allows for the use of a properties file with any name. Note that for Java applications, the wrapper binary is typically renamed to reflect the application it is launching.

`--propVar` *name=value*	Sets a property as though it were in the properties file, but at runtime instead.
	For example, assign an environment variable called `MYTEST` to a property in the `<wrapped application>.tra` file as follows:
	`java.start.class=%MYTEST%`
	You can then run the application as follows to set the value of the variable at runtime:
	`<Wrapped application> --debug --propVar MYTEST=<runtime value>`

`--run`	Runs the application as a console application, not as a service. This can be useful for debugging an application before installing it as a service.
`--help`	Lists the command line options and usage information.
`--version`	Displays the version of the wrapper utility.
`--pid`	Returns the process ID of the application to stdout in the form *application.processid=%d*

Table 43 Command-line Options (Microsoft Windows only)

Option	Description
`--install`	Installs the application as a service by registering it with the SCM and copying the contents of the properties file used into the Microsoft Windows Registry. Note that any subsequent changes to the properties file will *not* be propagated to the service unless it is first uninstalled and reinstalled.

Table 43 Command-line Options (Microsoft Windows only) (Cont'd)

Option	Description
`--uninstall`	First attempts to stop the service if it is currently running. After that, it requests the SCM to remove the application from the service database. Note that if the SCM was unable to stop the service, it will be removed the next time Windows is restarted. For this reason, it is possible to see the service listed in the Microsoft Windows Control Panel after having uninstalled it.
`--start`	Sends a message to the SCM to start the service registered with this service name. Obviously, the service must be installed first. This is a command line alternative to using the Microsoft Windows Control Panel Services applet to start a service.
`--stop`	Sends a message to the SCM to stop the service registered with this service name. Obviously, the service must be installed first. This is a command line alternative to using the Control Panel Services applet to stop a service.
`--restart`	Requests first a stop and then a start of the service.
`--query`	First checks whether the service has been installed. If so, queries the SCM as to the state of the service.

Appendix A SDK Programming Guidelines

This appendix gives programming guidelines for adapter developers.

Topics

General SDK Best Practices

This section gives some general Adapter SDK programming guidelines.

- Do not access any TIBCO Adapter SDK classes (for instance, `MTrace`) after `MApp.stop()` is called.

  After a custom adapter has called `MApp.stop()`, it can no longer access any of the objects that the class library itself created because `MApp` performs internal clean-up while executing the `stop()` method.

- It is not advisable to use direct calls to the TIBCO Rendezvous API within a custom adapter.

  Using direct calls makes the custom adapter less portable and more difficult to upgrade to newer versions of the SDK. However, in some cases using a direct call to a TIBCO Rendezvous API is the only workaround available. For example, an adapter design may rely on a TIBCO Rendezvous feature not available in the SDK API, such as RV message queue control or RVFT protocol.

- Use command-line properties files where appropriate.

  For example, you may want to have different properties files for starting the adapter in different modes. You can also obfuscate the properties file, which can make user names and passwords inaccessible. See Properties Files on page 39.

- Do not confuse out of sequence confirmation with selective confirmation.

  TIBCO Rendezvous RVCM allows you to confirm message out of sequence. This means you can confirm messages in any order such as 2, 3, 7, 5, then 6. This feature is often confused with selective confirmation, that is, confirming only certain messages.

  Assume you want to confirm message 1, 3 and 5 while leaving message 2 and 4 unconfirmed. Selective confirmation is not possible using the RVCM protocol. While you can confirm individual messages without implicitly confirming any unconfirmed gaps in the sequence, the listening CM transport holds the confirmation protocol message until all gaps are filled.

  For example, confirming message 57 does not implicitly confirm messages 55 and 56. Instead, the listening CM transport records the confirmation, but does not send it back to the sending transport until the program fills the gap by confirming messages 55 and 56, and then the listening transport sends the confirmations for messages 55 through 57.

- Make transport parameters flexible.

TIBCO Adapter SDK supports flexible transport parameters by keeping the adapter configuration, saved in the project repository, separate from the code. See Changing Endpoint Quality of Service on page 54.

- Do not access `MTree` instances directly.

  Instead of working with `MTree` instances, hand them to the deserializer. TIBCO Adapter SDK validates all deserialized ActiveEnterprise messages against metadata schema defined in the repository.

- Do not confuse the SDK properties file (and associated properties) with the TIBCO Wrapper properties file (and associated properties).

  While it is possible to define properties from each group in one file, how the properties are used differs. See Two Types of Properties on page 40.

- Separate the configuration from the code wherever possible.

  While constructors exist for SDK classes, it is better to define endpoints, sessions, etc. as part of the configuration.

MBusinessDocument and MAdvisoryDocument

`MBusinessDocument` is a predefined schema that provides context attributes. It allows a custom adapter to describe a business event in a standard format.

When a custom adapter sends a message to another application, it could send pure data. In many cases, however, it is useful to include context for the data. The context allows the receiving and sending application to share information that is not necessarily part of the data.

Typically, the custom adapter itself defines the context, that means each sending and receiving application need to agree on the format for the context. The `MBusinessDocument` class encapsulates a specification of such a context, which can be used to provide uniformity, for example, in integration scenarios where several adapters work together.

The `MBusinessDocument` architecture provides a flexible way to create user-defined attributes, without imposing a rigid structure. See the online API Reference documentation for details, including the methods for accessing and setting the attributes of `MBusinessDocument`.

The `MAdvisoryDocument` class is useful for providing context to debugging and performance analysis data. Whereas `MBusinessDocument` is useful for providing context to application data itself.

In TIBCO Designer, `MBusinessDocument` is defined under `AESchemas/ae/baseDocument`.

The predefined `MBusinessDocument`, `MDataSection`, and `MAdvisoryDocument` class schema are available for backward compatibility and should not be used for new development.

New development that would like to take advantage of `MBusinessDocument` should create custom class schema definition and specify `baseBusinessDocument` as the superclass. So, any subclass of `baseBusinessDocument` will inherit the predefined properties provided by `baseBusinessDocument`.

The same subclass use applies to `MAdvisoryDocument`.

Example

```
<class name="OrderBusinessDocument"
                                superclass="baseBusinessDocument">
 <attribute name="OrderId" type="String"/>
 <attribute name="Item" type="OrderLine" />
</class>
```

```
MInstance *pInstance =
            MDataFactory::createInstance("OrderBusinessDocument");
 MBusinessDocument *pDoc = new MBusinessDocument(pInstance);
```

Using Distributed Processes for Load Balancing

TIBCO Rendezvous Distributed Queue messaging (RVDQ also known as RVCMQ) can be used to implement message-level load balancing.

RVCMQ should be used if the adapter must process messages in the order sent or if the adapter must not process the same message twice (duplicate processing).

Distributed Queue Member

A distributed queue is a group of cooperating transport objects, each in a separate process. Each transport object is called a member. From the outside, a distributed queue appears as a single transport object; inside, the group members act in concert to process inbound task messages.

Each distributed queue member has two roles: as a worker and as a potential scheduler. In the worker role, members listen for task messages and process inbound task messages as assigned by the scheduler. To determine whether a member is a worker or scheduler, create an advisory listener on the RVCMQ advisory subject `_RV.INFO.RVCM.QUEUE.SCHEDULER.ACTIVE.>`. See Advisory Handling on page 138.

Worker Weight and Worker Tasks

RVCMQ provides message-level load distribution based on `Worker Weight` and `Worker Tasks`. Each worker in the workgroup can specify a different value for these parameters, thereby influencing how the work is distributed among workers. The value 0 is the reserved value.

If the parameters `Worker Weight` and `Worker Tasks` are set to the same value for all adapters, RVCMQ will distribute messages evenly. Depending on their capacities to process tasks, each adapter can specify a different value for each worker task. For example, if an adapter has *n* dispatchers, it can specify *n* tasks.

A thread pool design can also be combined with RVCMQ to increase the number of tasks a worker member can handle concurrently.

When using thread pools, make sure to increase the number of worker tasks correspondingly.

Scheduler

If the incoming rate of messages is very high and overwhelms the worker adapters, the adapter in which the RVCMQ scheduler lives will start taking up tasks itself. When a DQ member assumes a scheduler job, it overwrites its worker weight and worker task to 1.

You cannot avoid the likelihood of the scheduler being assigned work, but you can set a short completion time (for example, 1000 milliseconds) on the scheduler and disable automatic confirmation. This will reassign a certified message (task) assigned to the scheduler to an available worker one second later.

The actual completion time value to use depends on other DQ parameters and use cases, such as the number of tasks or threads per worker, the number of workers, the message rate, and so on. 1000 milliseconds is not appropriate for all cases.

APIs for Setting RVCMQ Backlog Size

TIBCO Adapter SDK provides the following APIs in the `MRvSession` class both for C++ and Java SDK to set the RVCMQ backlog size.

```
setTaskBacklogLimitInBytes()
setTaskBacklogLimitInMessages()
```

Connection Management

Protocol and database adapters usually require connection management using connection pools. A connection pool works as a system resource management facility that can help improve performance in an application interface to a backend system.

When a message arrives at an adapter, it is executed by a target application specific task. A pool of connections to the application should be available to an executing adapter task.

The pool of connections should be managed by a connection manager, which should have the following characteristics:

- Configurable connection pool size
- Configurable connection retry attempts and timeout
- Obtain and release connection

The connection pool is configured in TIBCO Designer and saved to the project repository. The code can then pick up the connection pool and work with it.

If the backend application processes data in a synchronous manner (blocking behavior), it is useful to maintain a request queue. Used in conjunction with a multithreaded design, the adapter can process multiple concurrent synchronous operations without loss of messages.

Security Considerations

The user management and domain monitoring components of TIBCO Administrator allow administrators to define an access control list (ACL) for repositories in the administration domain. See *TIBCO Administrator User's Guide* for more information.

The TIBCO Administrator administration server checks the users that access or invoke project repositories and ensure that they have the appropriate privileges defined in the ACL.

You are still ultimately responsible for securing the machines and file systems on which TIBCO products are running.

TIBCO Administrator Administration Server Access

File Repositories

Local file repositories must be secured using file system permissions. The same applies to a file-based repository managed by an administration server.

Local file repositories cannot be secured using tools provided by TIBCO because there is no enforcement mechanism available to stop a user from accessing a file, if the user has file access rights.

Server-Based Repositories

The read-only or read-write mode can be used for a server-based repository. The mode is specified in the `tibcoadmin.tra` file.

The read-only mode is only available if the server is operating in Load Balancing mode. All servers other than the `repo.master` operate as read-only. The master may switch its mode by changing the `tibcoadmin.tra` file directly and restarting, or using the Repository palette (need to provide an administration server password).

```
repo.state=READ_ONLY
repo.master=localhost
```

See the *TIBCO Administrator Server Configuration Guide* for more information about load balancing.

Password Obfuscation

A domain utility, which includes an obfuscation tool, is provided with TIBCO Runtime Agent. The obfuscation tool can be used to mask the password stored in a properties file.

The obfuscation tool is not an encryption tool.

For example, an adapter needs to log into the database, and the username and password are provided in a properties file. The password needs to be obfuscated so it is not recognizable to a casual user looking at the file.

1. In TIBCO Designer, define global variables for fields such as the username and password (`%%username%%`, `%%dbpassword%%`). Configure and save the adapter instance.

   These variables are user defined. `dbpassword` refers to a database the adapter wants to access.

2. Create a properties file and include the parameters `clientVar.username` and `clientVar.dbpassword`. Precede the parameters with a #. For example, `lientVar.username=#frog`.
3. Run the domain utility, (found under `tra/bin`) choose obfuscation as the task to perform, and supply the properties file.

   The obfuscation tool will create a new properties file of the same name and which contains the username and password in obfuscated form.

The adapter can be run with this properties file using the `-propFile` command line option.

Data Security

If data security for an adapter on the network is required, developers should consider using TIBCO Enterprise Message Service and implement the adapter to use SSL.

As an alternative, developers requiring the use of TIBCO Rendezvous can consider the transformation plug-in. See Transformation Plug-in on page 145.

TIBCO Rendezvous Programming Guidelines

This section lists the following guidelines for programming with TIBCO Rendezvous:

- Never Send Binary Data Buffers or Internal Structs, page 233
- Do Not Pass Local Values, page 234
- Do Not Send to Wildcard Subjects, page 234
- Control Message Sizes, page 234
- Avoid Flooding the Network, page 234
- Understand TIBCO Rendezvous Transport Parameters, page 235
- Establish Subject Naming Conventions, page 235
- RVCM Session Correspondent Name Restrictions, page 236
- Sessions and TIBCO Rendezvous Protocol, page 237

Never Send Binary Data Buffers or Internal Structs

Programs can exchange binary data buffers using data type `TIBRVMSG_OPAQUE`. The program is free to use any format and content within opaque data. However, extensive use of opaque data is not recommended.

For example, opaque buffers can contain data structures mapped by C language `structs`, but this technique couples the programs tightly to the data structure. If the `struct` definition in the sender is changed, it must also be changed in the listener, and vice versa.

Exchanging `structs` makes it difficult to introduce new, interacting programs in the future. Furthermore, exchanging internal `structs` makes it difficult for a program to interact with programs developed in other languages.

Binary data and internal structs are also platform dependent. Raw, binary data cannot be exchanged between programs running on machines that represent numbers or character strings with different formats.

Instead of binary buffers or structs, it is recommended to use TIBCO Rendezvous self-describing data to ease data exchange. Rendezvous data types span the most common atomic and array data types of most programming languages, and ActiveEnterprise wire format messages can emulate any struct or composite data type.

Do Not Pass Local Values

When exchanging structs or binary buffers, remember that many data types could be meaningless at the receiving end.

For example, a pointer is a memory address inside a particular computer. It has no meaning to any other program running on other computers. You must always send actual data by value rather than referencing it with a pointer.

Many opaque data structures or quantities are similarly meaningless outside of a particular program (for example, UNIX file descriptors). Do not send this kind of data to other programs.

Do Not Send to Wildcard Subjects

Although transports do not prevent you from sending messages to wildcard subject names, doing so may lead to unexpected behavior in other programs that share the network. It is invalid for certified delivery transports to send to wildcard subjects.

Control Message Sizes

Although the ability to exchange large data buffers is a feature of TIBCO Rendezvous software, it is recommended not to make messages too large.

For example, to exchange data up to 10,000 bytes, a single message is sufficient. But to send files that could be many megabytes in length, it would be better to use multiple send calls, perhaps one for each record, block or track.

Empirically determine the most efficient size for the prevailing network conditions. (The actual size limit is 64 MB, which is rarely an appropriate size.)

Avoid Flooding the Network

TIBCO Rendezvous software can support high throughput, but all computers and networks have limits. Do not write programs that might flood the network with message traffic. Other computers must filter all multicast messages, at least at the hardware level and sometimes at software levels (operating system or TIBCO Rendezvous daemon).

Do not create loops that repeatedly send messages without pausing between iterations. Pausing between messages helps leave sufficient network resources for other programs on the network. For example, if a program reads data from a local disk between network operations, it is unlikely to affect any other machines on a reasonably scaled and loaded network; the disk I/O between messages is a large enough pause.

Publishing programs can achieve high throughput rates by sending short bursts of messages punctuated by brief intervals. For example, structure the program as a timer callback function that sends a burst of messages each time its timer expires; adjust the timer interval and the number of messages per burst for optimal performance.

When a program sends messages faster than the network can accommodate them, its outbound message queue grows. When any outbound message waits in the outbound message queue for more than 5 seconds, TIBCO Rendezvous software presents a `CLIENT.FASTPRODUCER` warning advisory message. A program that receives this warning advisory message should slow its sending rate.

Understand TIBCO Rendezvous Transport Parameters

Transport creation calls accept two parameters that direct the transport to open two different kinds of sockets:

- The `service` parameter specifies a UDP or PGM service (also known as a UDP or PGM port). The transport opens a UDP or PGM socket to that network service.

  The TIBCO Rendezvous daemon processes uses the UDP or PGM service for communication with other TIBCO Rendezvous daemon processes across the network.

- The `daemon` parameter specifies a TCP port number. The transport opens a TCP socket to that port.

  Transport objects use the TCP port for communication between a client program and its TIBCO Rendezvous daemon (usually on the same host computer). This parameter corresponds to the `-listen` parameter of rvd.

These two types of socket are not interchangeable. Confusing the two leads to programming errors that are difficult to diagnose and fix. One source of this confusion is that the default TIBCO Rendezvous service (for TRPD daemons) is UDP service 7500, and the default daemon parameter is TCP socket 7500. Although these two numbers are the same, they specify different items.

Establish Subject Naming Conventions

Adapter configurations generated with the TIBCO Designer generic adapter palette creates predefined subject names based on global variables, which includes the adapter name, instance name, and schema class name.

It is recommended to keep and use the predefined subjects, and use a similar convention when defining your own subject.

If the predefined subject is not used, it is important to carefully plan the subject naming conventions for programs, and document them clearly for reference. Follow these guidelines:

- Plan naming conventions to reflect the logical structure of the data in the application domain.
- Study the programming examples in the `src/examples/` subdirectory.
- When designing naming conventions, think about the kinds of information that the programs will receive. Also think about the kinds of information that the program will ignore.
- Use a reasonably small number of levels in subject names. Four or five is usually sufficient. (TIBCO Rendezvous software allows many levels in subject names, but it is recommended to minimize the number of levels you actually use.)
- Avoid the use of spaces and special characters even where permitted by the TIBCO Rendezvous API. Such characters may cause trouble with various editors, browsers, and other tools.
- Keep subject names manageable and readable.
- Keep subject names short for maximum speed and message throughput.
- Allocate the maximum storage for subject names. Subject name length is artificially limited to 255 bytes so that programs can allocate name buffers with a reasonable size. To maximize code reusability, allocate 255 bytes for buffers that are receiving subject names, even if the program does not use long names. In C and C++, the 255 byte limit is defined by the constant `TIBRV_SUBJECT_MAX` in the TIBCO Rendezvous header files.
- Structure subject names so that subscribing programs can use wildcards effectively. Using wildcards is a powerful technique to filter inbound messages. Wildcards also offer a convenient way to subscribe to groups of subjects with a single listening call.

RVCM Session Correspondent Name Restrictions

Correspondent names have the same syntax as TIBCO Rendezvous subject names. For more information about the syntax of reusable names and practical advice on selecting a reusable name, see the *TIBCO Rendezvous Concepts*.

Invalid subject name examples:

- "News..Natural_Disasters.Flood" (null element)
- "WRONG." (null element)
- ".TRIPLE.WRONG.." (three null elements)

Sessions and TIBCO Rendezvous Protocol

When creating a publisher or subscriber, ensure that the appropriate session (transport) is available.

Associate a publisher or subscriber that uses RV as the protocol with any type of session, whether RVA (Java only), RV, RVCM, or RVCMQ (subscriber only).

A publisher or subscriber that uses RVCM as a protocol requires an RVCM session. A subscriber that uses RVCMQ as a protocol requires an RVCMQ session.

If you attempt to create an RVCM or RVCMQ subscriber and you only have an RV session, an error occurs.

C++ Utility Classes and Methods

A number of utilities classes and methods are available for building a custom adapter.

MList, MMap, MString, and MWString

The C++ SDK includes some templated utility classes that make SDK-based adapters portable across platforms. Custom adapters can use the following classes:

- `MString`—Templated string functionality.
- `MWString`—Templated string functionality for unicode (`M_UTF16BE`) strings.
- `MMap`—Templated map functionality.
- `MList`—Templated list functionality.

Custom adapters can use the SDK classes in conjunction with the C++ Standard Template Library (STL). However, SDK methods only use the SDK classes.

The downCast() Method

A number of SDK methods have as their return value type a class that is a superclass of the direct parent class of an object. For example, certain methods return pointers to `MComponent`, where the component returned is an instance of a subclass of `MComponent`. The `downCast()` method allows applications to safely cast the return value to the appropriate subclass so that the subclass methods can then be called.

Custom adapters usually use the `downCast()` method as in the following example:

```
MPublisher* pPublisher = MPublisher::downCast (pComp);
if (pPublisher)...
//can call MPublisher methods now...
```

Instead of throwing an exception, NULL is returned when the cast operation fails.

SDK Enumerators

This section applies only to the C++ SDK. Java programmers can use the Java native enumerator classes.

The C++ SDK offers enumerator classes for a number of its classes. Table 44 lists available C++ enumerators that are included in the API reference.

Table 44 C++ SDK Enumerators

Enumerator	Use
`MComponentEnumerator`	Extracts all components managed by one `MApp` instance.
`MTreeEnumerator`	Iterates through all (non-nested) nodes in an `MTree`.
`MPropertiesEnumerator`	Iterates through `MProperties` instances.
`MListEnumerator`, `MMapEnumerator`	Iterates through `MList` and `MMap` instances.
`MEnumerator`	Base enumerator.

SDK Types

The C++ SDK supports the type `Mboolean`, which is implemented as a `#define`. Its value can be either `Mtrue` or `Mfalse`. The type provides cross-platform boolean support.

JMS in Adapter SDK

This section explains the current JMS implementation in SDK and lists the major features of it.

JMS Implementation

The combined requirements of single-thread or multi-thread, and synchronous or asynchronous event dispatch resulted in the current Adapter SDK JMS support implementation.

When Adapter SDK started support for JMS, it provided a simple transport configuration switch (maintaining the existing `MSession`, `MPublisher`, `MSubscriber` object model) that allowed an adapter to change transports from Rendezvous to JMS.

To achieve this, Adapter SDK abstracted many of the differences between JMS and Rendezvous. One of the key differences is JMS session. Because event queue objects do not exist in the JMS world, `MJmsSession` only encapsulates `timemsConnection` and its parameters. On the other hand, `MRvSession` object encapsulates `tibrvTransport` and `tibrvQueue`.

To support `MApp::nextEvent()` and `MSession::nextEvent()` synchronous dispatch constructs, Adapter SDK uses synchronous event dispatch calls, `tibemsMsgConsumer_ReceiveNoWait()` and `tibemsMsgConsumer_ReceiveTimeout()`.

For multi-threaded `MDispatcher` asynchronous dispatch, Adapter SDK calls `tibemsMsgConsumer_ReceiveNoWait()` in a loop. If a message is retrieved, it calls `tibemsMsgConsumer_ReceiveNoWait()` again on the next iteration. Otherwise, it calls `tibemsMsgConsumer_ReceiveTimeout()` with a maximum timeout of 50 ms and minimum timeout of 1 ms. The upper limit is to maintain good response times, and the lower limit is to prevent high CPU usage.

Adapter SDK supports both single-threaded (default mode) and multi-threaded adapters (with `MApp::setMultiThreaded(true)`).

The single thread adapter requirement and synchronous event dispatch requirement excluded the use of the TIBCO Enterprise Message Service asynchronous event callback where a new thread is used to dispatch each incoming event.

Also, `MJmsSession` is associated with consumer and producer endpoint objects such that each gets its own `tibemsSession`, which can be mutex protected in the MT application dispatched using `MDispatcher`. `tibemsSession` needs to be protected because it is not a thread safe object per the JMS specification.

JMS Features

Increasing Throughput with JMS Queue Receivers

Having an equal number of consumers and dispatchers on a single queue destination increases throughput (as compared to one consumer and multiple dispatchers). This is analogous to creating a group of RVDQ subscribers to increase throughput.

The difference being RVDQ works best with queue members residing in separate running processes, while JMS queue receivers can reside within a single process. The downside is that message ordering is lost.

Multi-threading and Subscription Service

Adapter SDK manages construction and destruction of `tibemsSession` objects for an MT application. This allows multi-threaded consumption and production of messages by JMS adapter to work in a way similar to a Rendezvous adapter.

Configuring or creating multiple `MSubscriber` endpoints for a `MJmsSession` brings the similar effect of specifying multiple JMS sessions in a TIBCO ActiveMatrix BusinessWorks JMS receiver, which would create one consumer per JMS session. You can also combine this with `MDispatcher` multi-threading, which would dispatch when a consumer receives on one JMS session in each `MDispatcher` thread.

Specifying the Client ID for a JMS Session

If multiple consumers are connected to a JMS Session at the same time, you may need to identify which connection is yours. TIBCO Adapter SDK provides two approaches to specify a unique client identifier for a JMS session.

- Set the field named `Client ID` when configuring an EMS connection in TIBCO Designer.
- If you have created a JMS session in TIBCO Designer, you need to add a property named `tibco.jmsclientid.`*session_name* in the `.tra` file.

Filtering Messages by JMS Selectors

TIBCO Adapter SDK allows adapters to access JMS selectors in a JMS message. By doing so, adapters can filter JMS messages.

When creating an EMS Subscription Service, Adapter SDK automatically uses the value set for the field `MString messageSelector` in the class `MJmsEndpointSpec`.

In TIBCO Designer, there is a property `messageSelector` in the EMS endpoint configuration. Adapter SDK will use the value if the property is set.

Appendix B TIBCO Adapter Standards

This appendix outlines the requirements that are necessary for a custom adapter to meet before it is considered a standard adapter in the TIBCO ActiveEnterprise, with a special focus on integration with TIBCO ActiveMatrix BusinessWorks.

Topics

TIBCO Rendezvous License Ticket

An adapter must obtain an appropriate license for the messaging system being used (TIBCO Rendezvous or TIBCO Enterprise for JMS).

For TIBCO Rendezvous, it is possible to use an embedded license ticket, which can be built into the adapter's code using `MappProperties.setRvEmbeddedTicket()` and allows the adapter to run out of the box without a separate license ticket.

Contact the TIBCO representative for information on license tickets.

TIBCO Runtime Agent Considerations

TIBCO Runtime Agent provides shared libraries from TIBCO Adapter SDK and TIBCO Rendezvous, which are required to run the adapter. Adapters should be designed with a TIBCO Runtime Agent installation in mind because the end-user will not have the SDK. TIBCO Runtime Agent includes (but is not limited to) the following software:

- TIBCO Runtime Agent itself, which is the monitoring agent component
- TIBCO runtime libraries
- TIBCO Adapter SDK runtime libraries
- TIBCO Wrapper
- TIBCO Runtime agent (TIBCO Hawk) libraries
- TIBCO Administrator client library
- Third-party libraries (including Java Runtime Environment (JRE))

For an exact list of the software included with TIBCO Runtime Agent and the versions of that software, see the `readme.txt` in the `tra/`*version_num* folder.

Adapter Configuration Requirements

Adapters are configured using the TIBCO Designer generic palette.

Session Configuration

By default, session parameters are set to global variables (`%%RvService%%`, `%%RvNetwork%%`, `%%RvSession%%`) so that they can be changed at runtime through either global variable substitution or client variable substitution.

File Sink Configuration

By default, the log file is specified as `%%DirTrace%%/%%Deployment%%.%%InstanceId%%.log`.

The adapter must specify where `%%DirTrace%%` is either from the command line (using client variable substitution `-system:clientVar DirTrace=/some/log/dir`) or in the Global Variable definition itself. This means that all adapters should provide a launch script or use a wrapper launch executable that will specify this location.

Adapter Services Configuration

By default, the subject name is defined with global variable such as `%%Domain%%.%%Env%%.`*adaptername.resourcename*. Appropriate values must be defined to the `Domain` and `Env` global variables.

Adapter Services Requirements

One approach to enterprise application integration is to take a "services" view of the inter-application interfaces and operations.

Each interface is defined terms of the services one application provides to others, and the underlying technical details of how an application provides these services is deliberately hidden. One application needs not know anything about the other application's underlying data and functionality.

This approach allows an organization to reuse services throughout the enterprise, and insulates each application from underlying changes in other applications.

Publication Service and Subscription Service

Publication Service

A Publication Service generates an event in response to a target application event.

If the Publication Service sends a message in response to a target application event, the event is communicated to the publisher through notification mechanism of the target application. If there is no notification mechanism in the target application, the Publication Service must be configured to be polled using a timer.

Subscription Service

A Subscription Service converts the data it receives into an `MInstance` of the appropriate class that should match the subscriber service definition. The subscriber must then perform whatever target application task appropriate for the service, such as inserting the data into target application.

Compatibility with TIBCO ActiveMatrix BusinessWorks

For compatibility with TIBCO ActiveMatrix BusinessWorks, the adapter needs to provide at least the reliable message quality of service. The adapter should clearly define the available service and support a quality of service for each service.

Request Response Service and Request Response Invocation Service

Request-Response Service

Using the Request-Response Service, an adapter acts as a Request-Response server for ActiveEnterprise operations. Operations are defined in a schema in the repository. Input and output parameters and exceptions must be defined for each operation.

- A server is defined in the configuration and implemented as an AE operation implementation class (a class that extends `MOperationImpl`).
- A server may be synchronous or asynchronous (reply immediately or not). Whether a server is synchronous or asynchronous is transparent to the client.

Request-Response Invocation Service

Using the Request-Response Invocation Service, the adapter acts as a request-response client for ActiveEnterprise operations. A client is defined in the configuration and invokes an AE operation in a synchronous or asynchronous manner. A client invokes an operation by constructing an `MClientRequest` object and then calling `synchInvoke()`, `asynchInvoke()` or `onewayInvoke()`.

- Synchronous invocation blocks the operation and waits for a reply. Note that the application implementing a blocking method also requires concurrency in other aspects of the adapter. Consider a multithreaded design for such an adapter.
- Asynchronous invocation does not block and provides a callback function for listening for the reply.
- One-way invocation does not expect a reply.

Compatibility with TIBCO ActiveMatrix BusinessWorks

For compatibility with TIBCO ActiveMatrix BusinessWorkss, the adapter needs to provide at least the Operation Server Service if client/server operations are applicable and defined.

Integration Requirements

This section explains the requirements for an adapter to interact with ActiveEnterprise applications and requirements specific to TIBCO ActiveMatrix BusinessWorks.

Integration with ActiveEnterprise

Any adapter that wants to interact with ActiveEnterprise applications should meet the requirements listed in this section.

Standard Wire Format

An adapter interacting with TIBCO ActiveMatrix BusinessWorks should use one of the ActiveEnterprise wire formats (`aeRvMsg` or `aeXml`). If an adapter interacts with a second adapter that uses `rvMsg` format, it should use `rvMsg` as well.

Standard Tracing and Logging

A standard adapter should have at least one fileSink with errorRole, warnRole and infoRole defined. If a trace log file is not allowed, a network sink can be used to send trace message for logging by a separate application.

Each component will log the significant steps of a process to the infoRole, which should be turned ON even in normal production operation. A significant step should include the minimum amount of information required for normal operation monitoring. Consider the time duration of a step before defining a significant step.

If a process is taking a fraction of a second, multiple significant steps should not be traced in the log for each event.

Monitoring Integration with TIBCO Hawk

Adapters can implement a custom hawk microagent and methods or at least make use of the TIBCO Adapter SDK provided class microagent.

The class and standard microagent built into the SDK provides basic management methods. For example, the standard microagent can be used to inspect and change the adapter's trace sink and role assignment.

The adapter must have the ability to stop using TIBCO Hawk or other mechanism. An adapter can use the Adapter SDK provided standard method or implement its own. Optionally, a stop message subscriber can be implemented to gracefully shutdown the adapter.

A custom Hawk microagent and method should be implemented to allow dynamic runtime management of the adapter when applicable. Optionally, a custom Hawk method can be included to allow management of the adapter at runtime, such as the ability to toggle the custom debug trace role or off.

Integration with TIBCO ActiveMatrix BusinessWorks

In addition to the requirements listed under Integration with ActiveEnterprise on page 249, a number of requirements exist for running a custom adapter in a TIBCO administration domain as part of a TIBCO ActiveMatrix BusinessWorks integration project.

The *SDK_HOME*`/resourceKit/deployableAdapter` directory contains a ZAP adapter example program that was modified to run in an administration domain. The directory also contains step-by-step instructions on how to change ZAP adapter to make it compatible with TIBCO Administrator.

TIBCO Wrapper Integration

To provide a uniform adapter launch and shutdown interface, TIBCO Runtime Agent provides a TIBCO Wrapper executable for TIBCO Adapter SDK Java based applications and a TIBCO Wrapper library for TIBCO Adapter SDK C++ based applications.

See Chapter 13, TIBCO Wrapper Utility for details.

Integration with TIBCO Administrator

Generic Adapter Configuration

The following fields must be set to allow TIBCO Administrator to monitor a TIBCO ActiveMatrix BusinessWorks compliant adapter:

- **Logging Tab** The log file name in the palette should default to `%%DirTrace%%/%%Deployment%%.%%InstanceId%%.log`
- **Monitoring Tab** The standard microagent name should be `COM.TIBCO.ADAPTER.`*global_acronym*`.%%Deployment%%.%%InstanceId%%`

  Here *global_acronym* stands for the name that is provided in the domain utility while adding the adapter component to the domain. The checkbox for `Has Standard Micro Agent` should be checked, that is, the value should be `true`.

Class MicroAgent Name should be:

`COM.TIBCO.`*global_acronym*`.%%Deployment%%.%%InstanceId%%`

The value for `Has Class Micro Agent` should be `%%HawkEnabled%%`.

All runtime connection parameters for the adapter services should be defined as global variables and referred in the palette. Values should not be hard coded.

TIBCO Hawk Method Implementation

TIBCO ActiveMatrix BusinessWorks and its monitoring tool TIBCO Administrator has a different set of requirement for TIBCO Hawk methods, including the naming convention and code implementation that is needed for two TIBCO Adapter SDK standard microagent methods: `getAdapterServiceInformation()` and `getHostInformation()`. These two microagent methods allow TIBCO Administrator to obtain adapter runtime status (running, stopped, and so on), performance statistics, and adapter service information.

TIBCO Domain Registration

TIBCO Runtime Agent provides a domain registration utility tool under its `tibco/tra/`*version_num*`/tools/bin` directory. See the *TIBCO Administration Server Configuration Guide* for more information.

Appendix C TIBCO Adapter SDK Hawk Microagents and Methods

This appendix describes the available Hawk microagents and methods.

Topics

Overview

The TIBCO Adapter SDK includes two predefined TIBCO Hawk microagents:

- `COM.TIBCO.ADAPTER`. This microagent allows performing queries on all running adapters, regardless of their class/application.
- `COM.TIBCO.ADAPTER.`*zap* (where *zap* stands for the target software package with which the adapter is interfacing). This microagent allows performing queries on one class of adapter.

For more information about TIBCO Hawk, see the TIBCO Hawk documentation.

The methods in this appendix are documented as if they were C++ methods. In reality, the methods are accessible through the TIBCO Hawk Display and not language specific.

Available Microagents

lists the methods available for the adapter.

Table 45 Microagent Methods

Method	Description
COM.TIBCO.ADAPTER::activateTraceRole()	Activates a mapping of a role to a sink at runtime.
COM.TIBCO.ADAPTER::deactivateTraceRole()	Deactivates a mapping of a roles to sinks at runtime.
COM.TIBCO.ADAPTER::getAdapterServiceInformation()	Returns information about the services implemented by this adapter
COM.TIBCO.ADAPTER::getComponents()	Returns information about the currently active components.
COM.TIBCO.ADAPTER::getConfig()	Retrieves generic configuration information.
COM.TIBCO.ADAPTER::getConfigProperties()	Returns all attributes and elements for the given configuration property
COM.TIBCO.ADAPTER::getHostInformation()	Returns standard and extended application information set by using the `MHostInfo` class.
COM.TIBCO.ADAPTER::getRvConfig()	Returns information about TIBCO Rendezvous sessions defined by this adapter.
COM.TIBCO.ADAPTER::getRvQueueInfo()	Returns the size and priority for a TIBCO Rendezvous event queue in a Rendezvous session.
COM.TIBCO.ADAPTER::getServerLatency()	Returns MRpcServer latency.
COM.TIBCO.ADAPTER::getStatus()	Retrieves basic status information about the adapter.
COM.TIBCO.ADAPTER::getTraceSinks()	Returns information about sinks to which traces currently go.
COM.TIBCO.ADAPTER::getVersion()	Retrieves version information for the current application.
COM.TIBCO.ADAPTER::preRegisterListener()	Preregisters an anticipated listener.
COM.TIBCO.ADAPTER::reviewLedger()	Returns information retrieved from the ledger file of an RVCM session.
COM.TIBCO.ADAPTER::setTraceSinks()	Adds a role or changes the file limit of a previously specified sink.

Table 45 Microagent Methods (Cont'd)

Method	Description
COM.TIBCO.ADAPTER::stopApplicationInstance()	Stops the specified adapter by calling the internal `stop()` method.
COM.TIBCO.ADAPTER::unRegisterListener()	Unregisters a currently preregistered listener.

COM.TIBCO.ADAPTER::activateTraceRole()

Purpose Activates a mapping of a role to a sink at runtime.

This replaces the now deprecated `setTraceSink()` Hawk method.

Parameters

Name	Type	Description
Role Name	string	Name of the role to activate.
Sink Name	string	Name of a sink for which to activate the role.

Return

Name	Type	Description
None.		

COM.TIBCO.ADAPTER::deactivateTraceRole()

Purpose Deactivates a mapping of a roles to sinks at runtime.

Parameters

Name	Type	Description
Role Name	string	Name of the role to activate.
Sink Name	string	Name of a sink for which to activate the role.

Return

Name	Type	Description
None.		

COM.TIBCO.ADAPTER::getAdapterServiceInformation()

Purpose Returns information about the services implemented by the adapter.

The information is a summary of available adapter services. This information is set explicitly by the adapter writer using the `MAdapterServiceInformation` class.

Parameters

Name	Type	Description
Service Name	string	Name of the service from which to get information. Default is `ALL`.

Return

Name	Type	Description
Line	Integer	Sequential row number.
Service Name	string	Name of the Service.
Endpoint Name	string	Name of the endpoint used for this service.
Type	string	Type of the endpoint, for example, Publisher, Subscriber.
Quality of Service	string	QoS for the endpoint, for example, RV, RVCM.
Subject	string	Subject defined for this endpoint.
Class	string	Class associated with the subject.
Number of Messages	string	Number of messages processed for this endpoint.

COM.TIBCO.ADAPTER::getComponents()

Purpose Returns information about the currently active components such as publishers, subscribers, or timers.

Parameters

Name	Type	Description
Component Name	string	Name of the publisher or subscriber. Default is all.
Component Type	string	Any of `Publisher`, `Subscriber`, `Timer`, `Signal`, or `IO Descriptor`. Default value is `ALL`.

Returns

Returns	Type	Description
Instance ID	string	Name of this adapter instance.
Adapter Name	string	Name of the adapter.
Component Name	string	Name of the TIBCO Hawk component.
Component Type	string	The name of the TIBCO Adapter SDK class for this TIBCO Hawk component, such as `MPublisher`, `MSubscriber`, or `MIODescriptorSource`.
Session Name	string	Name of the TIBCO Rendezvous session.
Description	string	Information about this TIBCO Hawk component. For example, time interval, signal type, validating publisher (or subscriber) etc.

COM.TIBCO.ADAPTER::getConfig()

Purpose Retrieves generic configuration information. More specific configuration information is accessed through separate methods.

Parameters None.

Return

Name	Type	Description
Instance ID	string	Instance ID of this application.
Adapter Name	string	Name of the application.
Repository Connection	string	URL of the repository used for the configuration of this adapter instance.
Configuration URL	string	Location of the adapter instance description object inside the repository used for configuration.
Command	string	Command line arguments used to start the process.

COM.TIBCO.ADAPTER::getConfigProperties()

Purpose Returns all attributes and elements for the given configuration property.

Parameters

Name	Type	Description
Property	string	Name of the property for which elements (tags) and attributes are desired. For example, "pubsub/timer/interval".

Return

Name	Type	Description
Element Name	string	Repository directory for the property.
Attribute Name	string	Name of the repository object attribute (for example, `interval`).
Attribute Value	string	Value of the repository object attributes (for example, `2000`).
Line	integer	Line number in which this property is defined in the configuration file.

COM.TIBCO.ADAPTER::getHostInformation()

Purpose Returns standard and extended application information set by using the `MHostInfo` class.

Returns

Name	Type	Description
Name	string	Name of the property.
Value	string	Value of the property.

COM.TIBCO.ADAPTER::getRvConfig()

Purpose

Returns information about TIBCO Rendezvous sessions defined by the adapter.

Returns information about all currently defined sessions if no `sessionName` is provided.

Parameters

Name	Type	Description
Session Name	string	Name of the TIBCO Rendezvous session for which configuration is required (default is all).

Return

Name	Type	Description
Instance ID	string	The instance ID of this application.
Adapter Name	string	Name of the application.
Session Name	string	Name of the TIBCO Rendezvous session.
Service	string	Service group for this session.
Daemon	string	TIBCO Rendezvous daemon for this session.
Network	string	Network used by this session.
Session Type	string	The type of session; one of rv, rvcm, or rvcmq.
Certified Name	string	Name of this certified session.
Ledger File	string	Ledger file for RVCM session. Return the empty string for RV sessions that are not RVCM sessions.
CM Timeout	string	Timeout for this RVCM session. Return the empty string for RV sessions that are not RVCM sessions.

COM.TIBCO.ADAPTER::getRvQueueInfo()

Purpose Returns the size and priority for a TIBCO Rendezvous event queue in a Rendezvous session.

Parameters

Name	Type	Description
Session Name	string	Name of the TIBCO Rendezvous session for which information is required (default is all).

Return

Name	Type	Description
Session Name	string	Name of the TIBCO Rendezvous session.
Session Type	string	The type of session; one of rv, rvcm, or rvcmq.
TibrvQueue Size	integer	Event count in the Rendezvous session’s event queue.
TibrvQueue Priority	integer	Priority of the Rendezvous session’s event queue.

COM.TIBCO.ADAPTER::getServerLatency()

Purpose Returns MRpcServer latency.

Parameters

Name	Type	Description
Server Name	string	Name of the server (default is all).

Return

Name	Type	Description
Server Name	string	Activity name.
Number of Requests	string	Number of requests for this measurement.
Average Seconds per Request	string	Average seconds per request.
Max Seconds per Request	string	Maximum seconds per request.

COM.TIBCO.ADAPTER::getStatus()

Purpose Retrieves basic status information about the adapter. This information is fairly limited; for more information, additional methods are provided (COM.TIBCO.ADAPTER::getConfig() on page 261, COM.TIBCO.ADAPTER::getRvConfig(), etc.).

Parameters None.

Return

Name	Type	Description
Instance ID	string	Instance ID for this adapter instance.
Adapter Name	string	Name of the application.
Uptime	string	Number of seconds since startup.
Messages Received	integer	Number of TIBCO Rendezvous messages received.
Messages Sent	integer	Number of TIBCO Rendezvous messages published.
New Errors	integer	Number of errors since the last call to this method.
Total Errors	integer	Total number of errors since startup.
Process ID	integer	Process ID of the application.
Host	string	Name of host machine on which this application is running.

COM.TIBCO.ADAPTER::getTraceSinks()

Purpose Returns information about sinks to which traces currently go.

Parameters

Name	Type	Description
Sink Name	string	Name of the sink for which you need information. If no name is specified, information about all sinks is returned. Default is all.
Role Name	string	Name of the role for which you need information for the specified sink or sinks. Default is all.

Return

Name	Type	Description
Line	integer	Row number in which this property is defined in the configuration file.
Instance ID	string	Name of this adapter instance as a string.
Adapter Name	string	Name of the application for this sink.
Sink Name	string	Name of the sink.
Sink Type	string	Type of this sink. One of fileSink, rvSink, hawkSink, stderrSink.
Description	string	The sink description, for example, filename=<file>.
Roles	string	Roles this sink supports, as a string. For example "warning, error, debug".

COM.TIBCO.ADAPTER::getVersion()

Purpose Retrieves version information for the current application. Two lines may be returned, one for the SDK, the other for the application.

Parameters None

Return

Name	Type	Description
Instance ID	string	The instance ID as a string, for example, "finance".
Adapter Name	string	Name of the application as a string (for example, "r3").
Version	string	Version number as a string (for example, "2.0").

COM.TIBCO.ADAPTER::preRegisterListener()

Purpose Preregisters an anticipated listener.

Some sending applications can anticipate requests for certified delivery even before the listening applications start running. In such situations, the sender can preregister listeners, so TIBCO Rendezvous software begins storing outbound messages in the sender's ledger. If the listening correspondent requires old messages, it receives the backlogged messages when it requests certified deliver.

Parameters

Name	Type	Description
Listener Session Name	string	Name of the listener to preregister.
Publisher Name	string	Name of the component for which the listener should be preregistered.

Return This method returns true if the listener was preregistered successfully, false otherwise.

COM.TIBCO.ADAPTER::reviewLedger()

Purpose Returns information retrieved from the ledger file of an RVCM session.

Parameters

Name	Type	Description
Session Name	string	Name of the TIBCO Rendezvous session for which ledger information is desired (default is all).
Subject	string	Name of the subject for which ledger information is desired.

Return

Name	Type	Description
Session Name	string	Name of the RVCM session to which this information application.
Subject	string	Subject for this session.
Last Sent Message	integer	Sequence number of the most recently sent message with this subject name.
Total Messages	string	Total number of pending messages with this subject name.
Total Size	integer	Total storage (in bytes) occupied by all pending messages with this subject name. If the ledger contains ten messages with this subject name, then this field sums the storage space over all of them.
Listener Session Name	string	Within each listener submessage, the Listener Name field contains the name of the delivery-tracking listener.
Last Confirmed	string	Within each listener submessage, the Last Confirmed field contains the sequence number of the last message for which this listener confirmed delivery.
Unacknowledged Messages	integer	Number of messages pending for this listener. The value is computed by subtracting the last sent sequence number from the last acknowledged sequence number.
Line	integer	Row number in ledger file.

COM.TIBCO.ADAPTER::setTraceSinks()

Purpose Adds a role or changes the file limit of a previously specified sink.

Parameters

Name	Type	Description
Sink Name	string	Name of the sink for which you want to add a role or change the file limit.
Role Name	string	Name of the role you want to add to this sink (warning, error, debug, or user defined). Default is all.
File Size	integer	Maximum file size for this sink in bytes. This parameter is ignored if the sink specified by sinkName is not a file sink.

Return Return OK if successful or an error if not successful.

COM.TIBCO.ADAPTER::stopApplicationInstance()

Purpose	Stops the specified adapter by calling the internal `stop()` method.
Return	This method returns OK if successful or an error if not successful.

COM.TIBCO.ADAPTER::unRegisterListener()

Purpose Unregisters a currently preregistered listener.

Parameters

Name	Type	Description
Listener Session Name	string	Name of the listener to unregister.
Publisher Name	string	Name of the component for which the listener should be preregistered.

Return This method returns true if the listener was unregistered successfully, false otherwise.

Index