software AG

webMethods EntireX

EntireX Java Wrapper

Version 10.5

October 2019

WEBMETHODS

This document applies to webMethods EntireX Version 10.5 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: EXX-EEXXJAVAWRAPPER-105-20220422

Table of Contents

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at **https://documentation.softwareag.com**.

In addition, you can also access the cloud product documentation via **https://www.softwareag.cloud**. Navigate to the desired product and then, depending on your solution, go to “Developer Center”, “User Center” or “Documentation”.

Product Training

You can find helpful product training material on our Learning Portal at **https://knowledge.softwareag.com**.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at **https://tech-community.softwareag.com**. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at **https://github.com/softwareag** and **https://hub.docker.com/publishers/softwareag** and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at **https://empower.softwareag.com**. Many services on this portal require that you have an account. If you do not yet have one, you can request it at **https://empower.softwareag.com/register**. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

I Introduction to the Java Wrapper

The EntireX Java Wrapper provides access to EntireX RPC-based components from Java applications. With EntireX Java RPC you can develop both client and server applications written in Java. Java applets can also be used as EntireX RPC clients.

To use the Java Wrapper, the IDL file must be in a Java project.

The Java Wrapper uses the properties of an IDL file to

- use a source folder (the folder that the Java builder uses to compile Java source classes)
- make the classes public
- extend a custom class for the RPC client
- put the client and the tester in a client package
- put the server in a server package
- set a superclass to be extended by all the bean-compliant client-generated classes

The Java Wrapper generates multiple Java sources from an IDL file. If there is a related client-side mapping file (Natural | COBOL), this is also used (internally). The following sources can be generated:

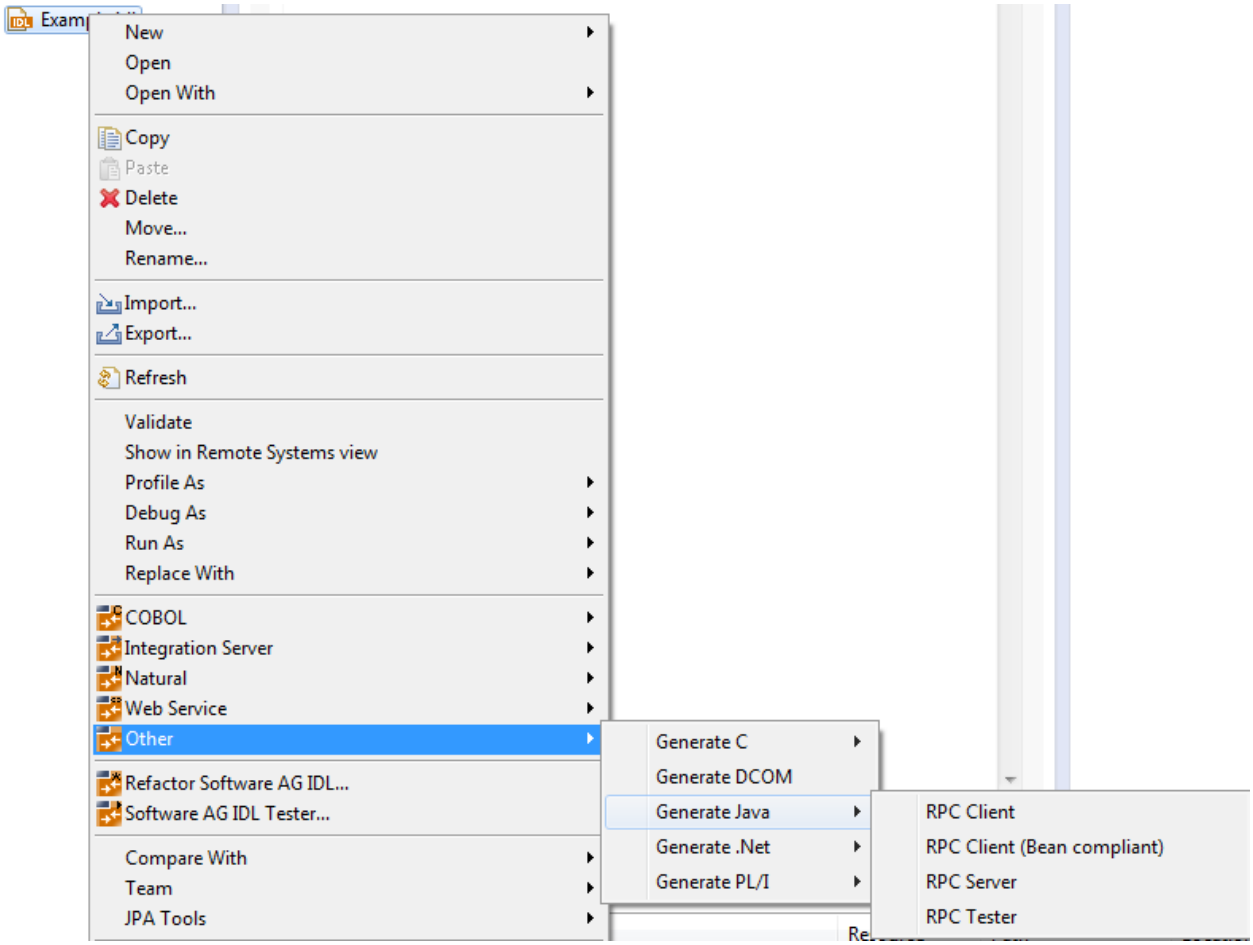
- RPC client
- RPC client (Bean-compliant)
- RPC server
- RPC tester

2 Using the Java Wrapper

Generating Java Sources

Select an IDL File

To generate a Java source, select an IDL file and, using the context menu, choose **Other > Generate Java > RPC Client**, **RPC Client (Bean compliant)**, **RPC Server** or **RPC Tester**.

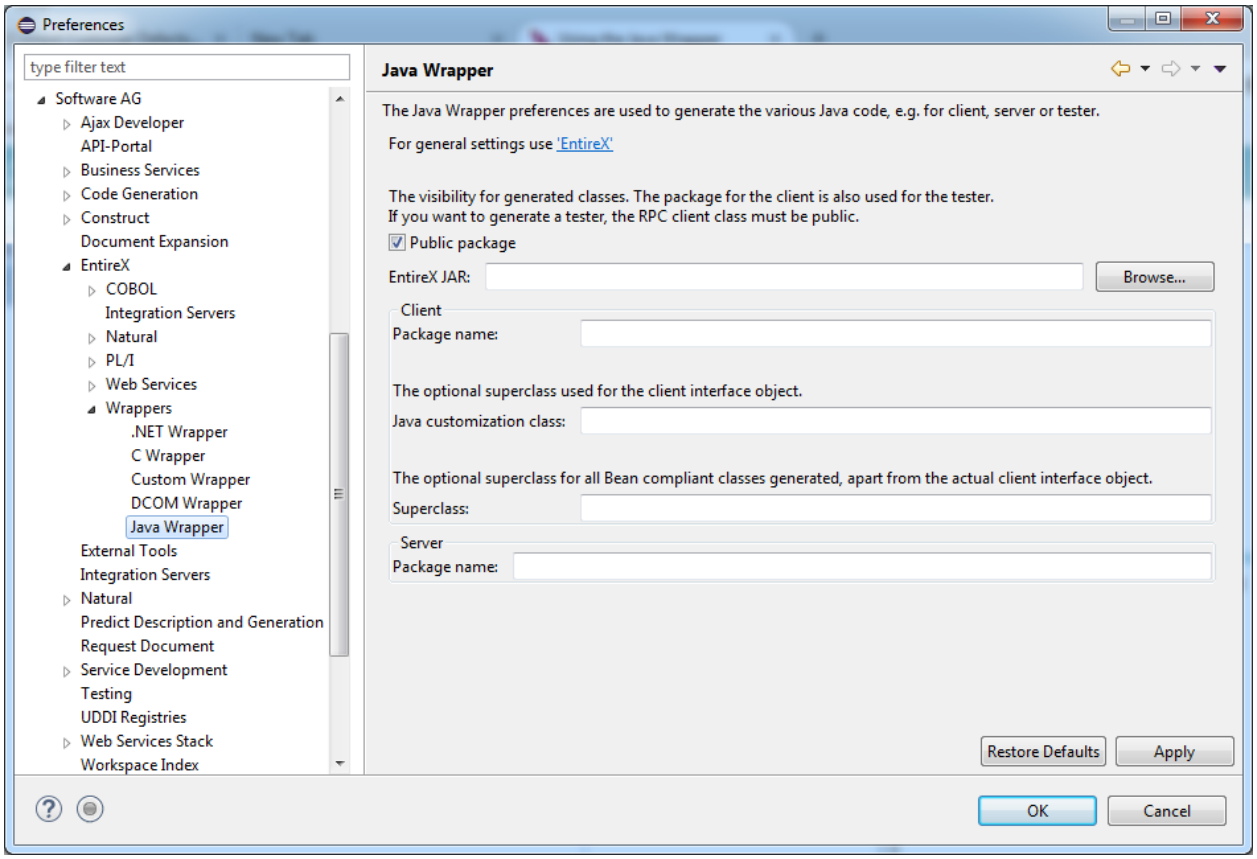


In addition to the standard commands of Eclipse, the context menu of a Java file contains a group of commands for the Java Wrapper.

Command	Description
RPC Client	Generates a Java client class.
RPC Client (Bean compliant)	Generates Java (client) classes instead of inner classes. There is one client class generated for each library in the Software AG IDL file.
RPC Server	Generates a Java server class and a server skeleton for your own implementation.
RPC Tester	Generates a client test program.

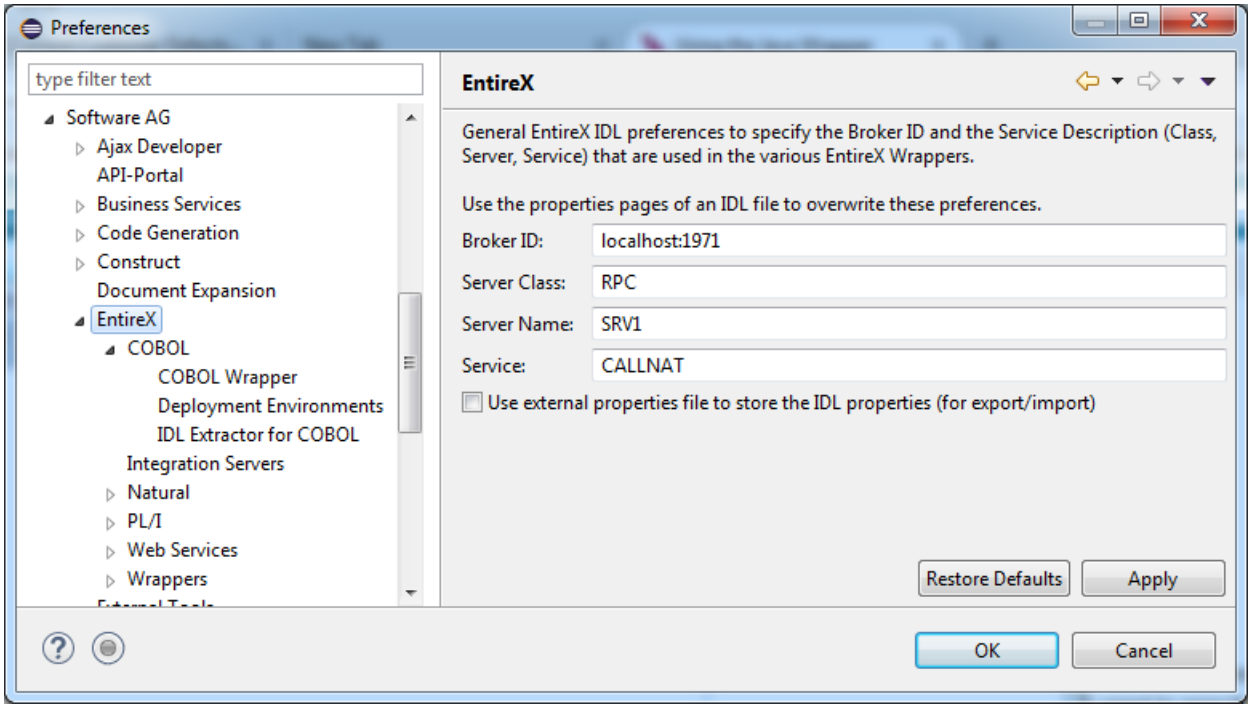
Important: If the IDL file is in a Java project, the Java Wrapper uses the project to compile the Java files. If the IDL file is in a simple project, the Java files are generated, but not compiled.

Preferences



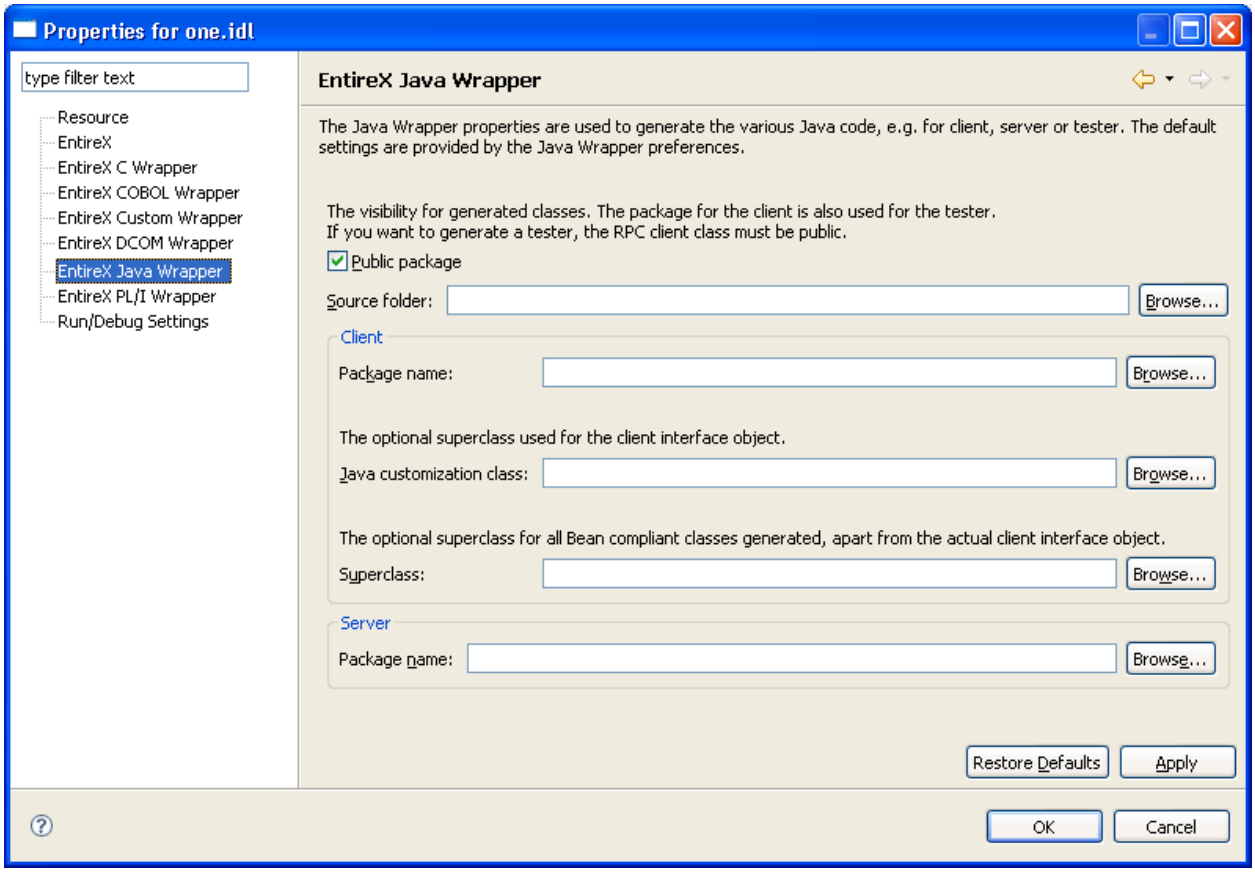
In general, the preferences of the Java Wrapper are used to set the Customization Class, the RPC client package name, and the RPC server package name; see *Configuring the RPC Server*. The package for the client is also used for the tester. If you want to generate a tester, the RPC client class must be public. The **Superclass** field is used to specify an extension class for all Bean-compliant generated classes apart from the actual client interface object.

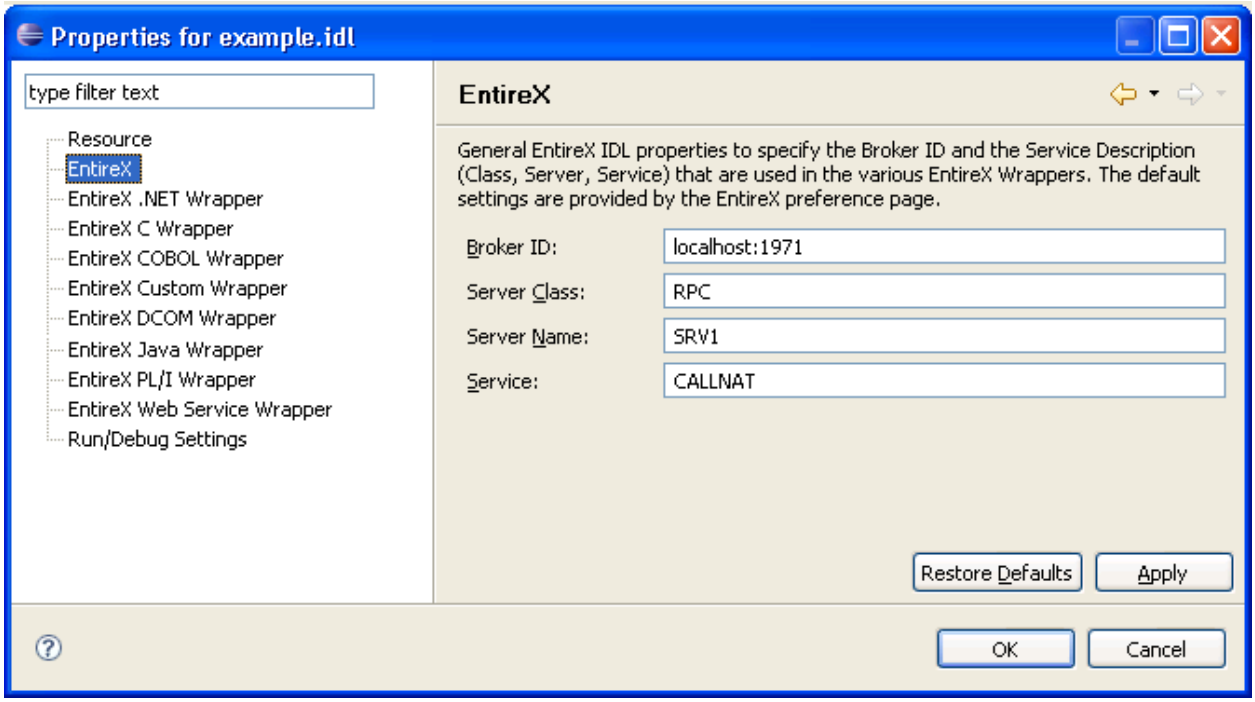
To set the broker ID and the server address for all new IDL files in the workspace, use the preference page “EntireX”.



Properties

For the settings of an individual IDL file, use the properties of this file. The property pages include the same fields to set as the preference pages. In addition, the property page of the Java Wrapper includes the project-specific setting of the source folder. This is the package root of the generated files.


Properties for one.idl
type filter text
Resource
EntireX
EntireX C Wrapper
EntireX COBOL Wrapper
EntireX Custom Wrapper
EntireX DCOM Wrapper
EntireX Java Wrapper
EntireX PL/I Wrapper
Run/Debug Settings
EntireX Java Wrapper
The Java Wrapper properties are used to generate the various Java code, e.g. for client, server or tester. The default settings are provided by the Java Wrapper preferences.
The visibility for generated classes. The package for the client is also used for the tester.
If you want to generate a tester, the RPC client class must be public.
Public package
Source folder:
Browse...
Client
Package name:
Browse...
The optional superclass used for the client interface object.
Java customization class:
Browse...
The optional superclass for all Bean compliant classes generated, apart from the actual client interface object.
Superclass:
Browse...
Server
Package name:
Browse...
Restore Defaults
Apply
OK
Cancel


Properties for example.idl
type filter text
Resource
EntireX
EntireX .NET Wrapper
EntireX C Wrapper
EntireX COBOL Wrapper
EntireX Custom Wrapper
EntireX DCOM Wrapper
EntireX Java Wrapper
EntireX PL/I Wrapper
EntireX Web Service Wrapper
Run/Debug Settings
EntireX
General EntireX IDL properties to specify the Broker ID and the Service Description (Class, Server, Service) that are used in the various EntireX Wrappers. The default settings are provided by the EntireX preference page.
Broker ID:
localhost:1971
Server Class:
RPC
Server Name:
SRV1
Service:
CALLNAT
Restore Defaults
Apply
OK
Cancel

Starting the IDL Tester

There are two alternatives for starting the EntireX IDL Tester:

- **From the Context Menu**
  This is the preferred method. In the context menu of the IDL file, choose **Software AG IDL Tester...**. A dialog appears for choosing the program to test.

  The IDL Tester is generated and launched as a separate Java Application. See *EntireX IDL Tester* for more details.

- **From Generated Test Program**
  To start the IDL Tester, select the generated test program in the Navigator or Package Explorer and choose **Run** from the context menu or toolbar.

  The IDL Tester is started as a separate application. See ***Using the IDL Tester***.

Generating a Java Client Interface Object

To generate a Java client interface object

1 In the Navigator view or in the Package Explorer, select the Software AG IDL file.

2 From the context menu, choose **Other > Generate Java > RPC Client**.

  This starts the generation of the Java source. The Java source files are written to the source folder of the IDL file. The source folder is set in the properties of the IDL file.

This starts the generation and compiles the generated Java sources. The Java source files and the class files are written to the directory of the IDL file.

File	Description
<Library name>.java	The Java source code of the generated client interface object. The library name is used to build the file name and the class name. Do not change this file.

If more than one library is defined in the IDL file, separate client interface object files will be generated for each library.

Generating a Java Client Interface Object without inner Classes (Bean-compliant)

When using the Java Wrapper to generate an RPC client (Bean-compliant), the resulting client interface object contains no inner classes. Instead, there will be separate classes generated for each structure within the IDL file.

Note: A superclass to be extended by all the newly generated classes can be specified in the Java Wrapper *Preferences* and in the *Properties* of the IDL file.

To generate a Java client interface object (Bean-compliant)

1 Select an IDL file.

2 From the context menu, choose **Other > Generate Java > RPC Client (Bean-compliant)**.

As a result, the generation of the Java source is started. The Java source files are written to the source folder of the IDL file and the generated Java sources are compiled.

Note: The source folder can be specified in the Java Wrapper *Preferences* and in the *Properties* of the IDL file.

The Java source files and class files are written to the directory of the IDL file. The following table gives a short description:

File	Description
<Library name>.java	The Java source code of the generated client interface object. The library name is used to build the file name and the class name. Do not change this file.
<Structure name>.java	A Java class is generated for each structure and group within the input IDL file(s).

Note: If more than one library is defined in the IDL file, separate client interface object files will be generated for each library.

Generating a Java Server Interface Object

To generate a Java server interface object

1 In the Navigator view or the Package Explorer, select the Software AG IDL file.

2 From the context menu, choose **Other > Generate Java > RPC Server**.

The Java Wrapper produces the following files for the server interface object in the source folder of the IDL file.

File	Description
<Library name>Stub.java	The Java source code of the generated server interface object. The library name followed by "Stub" is used to build the file name. Do not change this file.
<Library name>Server.java	A Java source file that contains a server skeleton. This is a complete Java class that can be compiled. It contains all methods the server has to implement. Add your application-specific coding in the places marked with the `// insert your application specific code here` comment. The library name followed by "Server" is used to build the file name. If this file exists, it will not be generated.
Abstract<Library name>Server.java	A Java source file that contains the generated part of the server as an abstract class. The server skeleton *<Library name>Server.java* extends this class and contains the application-specific code. Separating the generated code and the application-specific code simplifies re-generation of the RPC server.

If more than one library is defined in the IDL file, separate server interface object files will be generated for each library. The server package name is used as the package name in the generated server files. The server package is part of the Java Wrapper properties of the IDL file. At runtime, configure the server packages in the RPC Server for Java configuration. The RPC Server for Java uses the library name (which is part of the RPC request from the client) to dynamically load a class named *<Library name>Stub.class*. The RPC server searches for this server interface object class as well as the server class using the actual classpath.

Using the IDL Tester

The client test program is an easy-to-use utility to check whether the remote call works. The client test program supports most of the data types and features of the IDL.

If there is no client interface object already defined, the IDL Tester will generate a Bean-compliant client interface object. However, if there is a previously generated client interface object, it will not be overwritten, regardless if it is Bean-compliant or not.

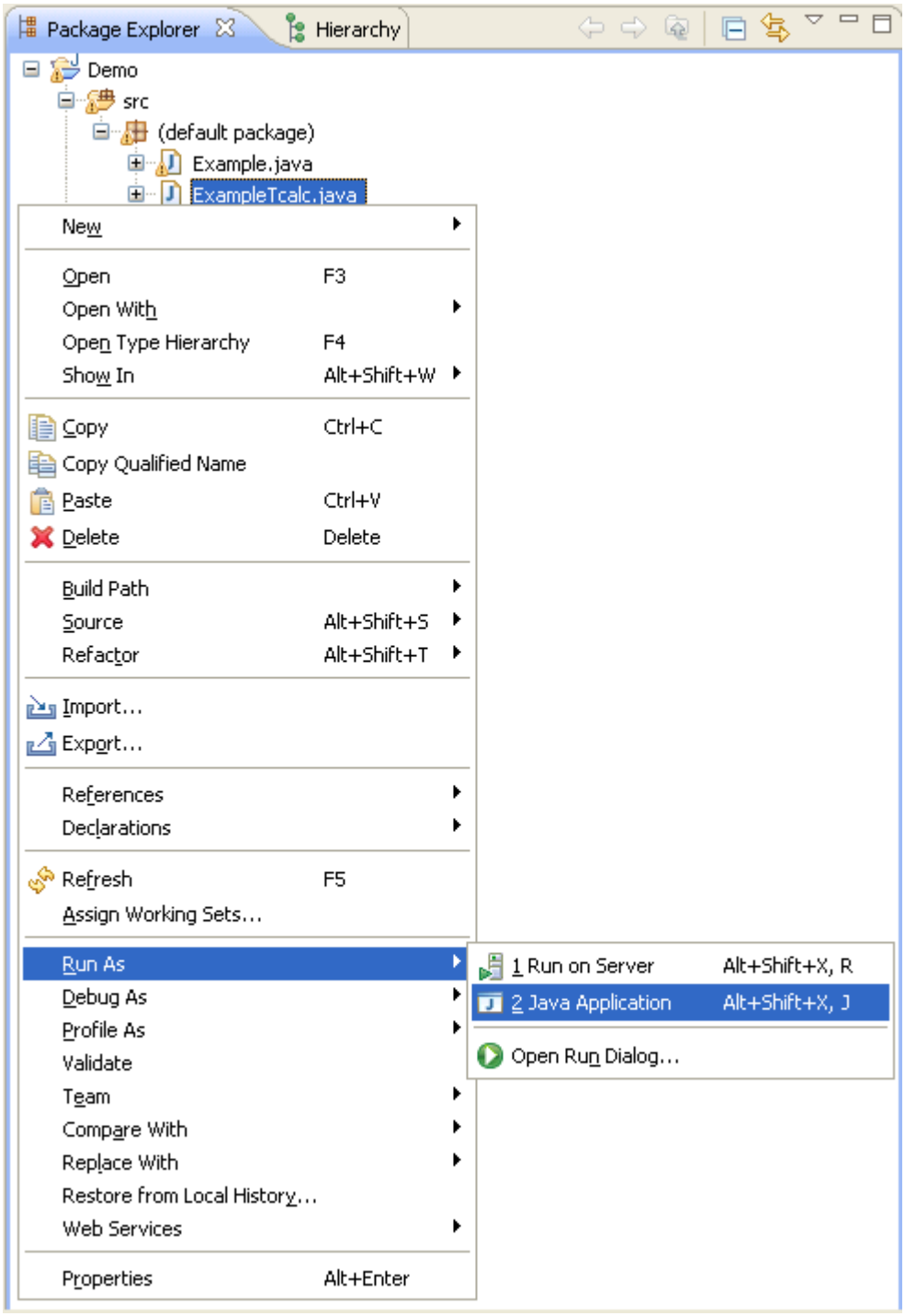
There are two alternatives for generating and running the standard client test program:

- From the context menu of an IDL file. This is the preferred method. See *EntireX IDL Tester* in the Designer documentation.
- From the context menu, using **Other > Generate Java > RPC Tester**. See below.

This section covers the following topics:

Calling the IDL Tester using Other > Generate Java > RPC Tester

1. In the Navigator view or in the Package Explorer, select the Software AG IDL file.
2. From the context menu, choose **Other > Generate Java > RPC Tester**. For each program in the IDL file, one class with the name `<Library name>T<program name>.java` is generated. The class `<Library name>T<program name>` can be started as a standalone Java application.



3. In the Navigator view or the Package Explorer, select the file *<Library name>T<program name>.java* and choose **Run As** from the context menu or **Run...** from the **Run** menu. This creates a launch configuration and starts the tester.

See *EntireX IDL Tester* in the Designer documentation for more information.

Using the IDL Tester in Batch Mode

To start the Tester in Batch mode

- Enter the following command

```
java -classpath <your classpath> <library>T<program> -batch
```

where `<your classpath>` contains the class of the RPC tester and the file *entirex.jar*
`<library>` is the name of the library and
`<program>` is the name of the program

For the delivered *example.idl*, the following RPC testers are provided: *ExampleTcalc*, *ExampleThello*, *ExampleTpower*.

An RPC is executed with the default values.

If you add `-both` instead of `-batch`, the GUI of the tester is opened, but the messages and parameter values are written to SYSOUT, too.

To change the broker ID, use `-b <broker id>`. To change the server address, use `-s <class/server/service>`, for example:

```
java ExampleTcalc -b localhost:1971 -s RPC/SRV1/CALLNAT + 3 5
```

To modify the default values

- In the command line add the parameters to the commands.

They will be assigned to the input values one after the other. Enter, for example `java ExampleTcalc + 3 5` to calculate 8.

3 Using the Java Wrapper in Command-line Mode

See *Using EntireX in the Designer Command-line Mode* for the general command-line syntax.

Command-line Options

Note: The commands `-java:allbeancompliant`, `-java:tester` and `-java:clientbeancompliant` commands will generate a Bean-compliant Java client. To generate client with inner classes (the old way) use `-java:client` or `-java:all` commands.

Task	Command	Option	Description
Generate all Java source files for the specified IDL file(s).	`-java:all`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the JavaBean-compliant Java client(s) for the specified IDL file(s).	`-java:allbeancompliant`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java client(s) for the specified IDL file(s).	`-java:client`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-public`	Generate a public wrapper class.
		`-sourcefolder`	The folder for the generated classes.
Generate the JavaBean-compliant Java client(s) for the specified IDL file(s).	`-java:clientbeancompliant`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.

Task	Command	Option	Description
		`-public`	Generate a public wrapper class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java server(s) for the specified IDL file(s).	`-java:server`	`-help`	Display this usage message.
		`-serverpackage`	The server package name for the server interface object class.
		`-sourcefolder`	The folder for the generated classes.
Generate the Java client(s) and tester(s) for the specified IDL file(s).	`-java:tester`	`-clientpackage`	The client package name for the wrapper class.
		`-customclass`	A non-default superclass of the wrapper class.
		`-help`	Display this usage message.
		`-sourcefolder`	The folder for the generated classes.

Example

```
<workbench> -java:client /Demo/Example.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

where *<workbench>* is a placeholder for the actual EntireX design-time starter as described under *Using EntireX in the Designer Command-line Mode*.

The name of the IDL file and the source folder include the project name. In the example, the project *Demo* is used. If the IDL file name describes a file inside the Eclipse workspace, the name is case-sensitive.

If the first part of the IDL file name is not a project name in the current workspace, the IDL file name is used as a file name in the file system. Thus, the IDL files do not need to be part of an Eclipse project.

If the source folder does not exist in the workspace but the first part describing the project exists, the source folder is created.

If the IDL file is located outside the Eclipse workspace, the source folder is also a folder in the file system.

Status and processing messages are written to standard output (stdout), which is normally set to the executing shell window.

Further Examples

Windows

- **Example 1:**

```
<workbench> -java:client C:\Temp\example.idl -sourcefolder src -clientpackage ↩
com.client
```

Uses the IDL file at *C:\Temp\example.idl* and generates the Java source files to the subfolder *src\com\client* of the current working directory.

Output to standard output:

```
Using workspace file:/C:/myWorkspace/.
Processing IDL file C:\Temp\example.idl
Writing to file src/com/client/Example.java.
Exit value: 0
```

- **Example 2:**

```
<workbench> -java:client C:\Temp\*idl -sourcefolder src -clientpackage com.client
```

Generates Java clients for all IDL files in *C:\Temp*.

- **Example 3:**

```
<workbench> -java:client C:\Temp\example.idl -sourcefolder C:\Temp\src ↩
-clientpackage com.client
```

Uses the IDL file at *C:\Temp\example.idl* and generates the Java source files to *C:\Temp\src\com\client*.

- **Example 4:**

```
<workbench> -java:client C:/Temp/example.idl -sourcefolder C:/Temp/src ↩
-clientpackage com.client
```

The same as above. Both slashes and backslashes are permitted.

- **Example 5:**

```
<workbench> -java:client -help
```

or

```
<workbench> -help -java:client
```

Both show a short help for the Java client wrapper.

Linux

- **Example 1:**

```
<workbench> -java:client /Demo/Example.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

If the project *Demo* exists in the workspace and *Example.idl* exists in this project, this file is used. Otherwise, */Demo/Example.idl* is used from file system.

- **Example 2:**

```
<workbench> -java:client /Demo/*.idl -sourcefolder /Demo/src1 -clientpackage ↩
com.client
```

Generates Java clients for all IDL files in project *Demo* (or in folder */Demo* if the project does not exist). The generated files are in */Demo/src1/com/client*.

- **Example 3:**

```
<workbench> -java:client -help
```

or

```
<workbench> -help -java:client
```

Both show a short help for the Java client wrapper.

4 Software AG IDL to Java Mapping

Mapping IDL Data Types to Java Data Types

In the table below, the following metasymbols and informal terms are used for the IDL.

- The metasymbols "[" and "]" enclose optional lexical entities.
- The informal term *number* (or in some cases *number1*.*number2*) is a sequence of numeric characters, for example 123.

Software AG IDL	Description	Java Data Types	Note
A*number*	Alphanumeric	`String`	1, 3
AV	Alphanumeric variable length	`String`	
AV[*number*]	Alphanumeric variable length with maximum length	`String`	1
B*number*	Binary	`byte[]`	1, 6
BV	Binary variable length	`byte[]`	
BV[*number*]	Binary variable length with maximum length	`byte[]`	1
D	Date	`java.util.Date`	5
F4	Floating point (small)	`float`	2
F8	Floating point (large)	`double`	2
I1	Integer (small)	`byte`	
I2	Integer (medium)	`short`	
I4	Integer (large)	`int`	
K*number*	Kanji	`String`	1
KV	Kanji variable length	`String`	
KV[*number*]	Kanji variable length with maximum length	`String`	1
L	Logical	`boolean`	
N*number1*[.*number2*]	Unpacked decimal	`java.math.BigDecimal`	4
NU*number1*[.*number2*]	Unpacked decimal unsigned	`java.math.BigDecimal`	4
P*number1*[.*number2*]	Packed decimal	`java.math.BigDecimal`	4
PU*number1*[.*number2*]	Packed decimal unsigned	`java.math.BigDecimal`	4
T	Time	`java.util.Date`	5
U*number*	Unicode	`String`	7
UV	Unicode variable length	`String`	7
UV*number*	Unicode variable length with maximum length	`String`	7

Notes:

1. The field length is given in bytes.

2. If floating-point data types are used, rounding errors can occur. Therefore, the values of sender and receiver might differ slightly.
3. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for type A, a blank string will be used.
4. For Java, the total number of digits (`number1+number2`) is 99, which is the maximum that EntireX supports. See *IDL Data Types*.

   If you connect two endpoints, the total number of digits used must be lower or equal than the maxima of both endpoints. For the supported total number of digits for endpoints, see the notes under data types N, NU, P and PU in section *Mapping Software AG IDL Data Types* in the respective Wrapper or language-specific documentation.

   If you use the value null (null pointer) for direction `IN` (for `IN` and `INOUT` parameters), the value 0 (or 0.0) will be sent. See ***Mapping the Direction Attributes In, Out, InOut***.
5. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for types D/T, the current date/time will be used. You change this with the property `entirex.marshal.date`. Setting `entirex.marshal.date=null` will map the value null to the invalid date 0000-01-01 of the RPC marshalling. This is the invalid date value in Natural, too. With this setting the invalid date as an output parameter will be mapped to null. The default is to map the invalid date to 0001-01-01.
6. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters) for type B, all binary values will be set to zero.
7. The length is given in 2-byte Unicode code units following the Unicode standard UTF-16. The maximum length is 805306367 code units.

Please note also hints and restrictions on the Software AG IDL data types valid for all programming language bindings. See *IDL Data Types*.

Mapping Library Name and Alias

The library name as specified in the IDL file is sent from a client to the server. Special characters are not replaced. The library alias is not sent to the server.

In the RPC server, the IDL library name sent may be used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The library name as given in the library definition of the IDL file is mapped to the class name of the generated Java classes. See `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation. For the server interface object, the names of the class are composed as *`library name`* Interface Object and *`library name`* Server. For the client interface object, no suffix is appended. When the class names are built, the library name is capitalized to match Java naming conventions.

The special characters '#' and '-' in the library name are replaced by the character '_'.

If there is an alias for the library name in the `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation, this alias is used as is to form the client class name. Therefore, this alias must be a valid Java class name. On the server side, the alias is used as is to form the class name of the server class.

Example:

- library name Hu#G-O is converted to Hu_g_o

Mapping Program Name and Alias

The program name is sent from a client to the server. Special characters are not replaced. The program alias is not sent to the server.

In the RPC server, the IDL program name sent is used to locate the target server. See *Locating and Calling the Target Server* in the platform-specific administration or RPC server documentation.

The program name as given in the `program-definition` under *Software AG IDL Grammar* in the IDL Editor documentation of the IDL file is mapped to method names within the generated Java classes. To match Java naming conventions the program name is converted to lowercase.

The special characters '#' and '-' in the program name are replaced by the character '_'.

If there is an alias for the program name in the `program-definition` under *Software AG IDL Grammar* in the IDL Editor documentation, this alias is used as is for the method name. Therefore, this alias must be a valid Java method name. On the server side, the alias is used as is for the method name in the server class.

Mapping Parameter Names

The parameter names are mapped to fields inside the classes (see ***Mapping the Direction Attributes In, Out, InOut***).

Example:

- parameter name `Hu#G-O` is converted to `hu_g_o`

Mapping Fixed and Unbounded Arrays

Arrays in the IDL file are mapped to Java arrays. If an array value does not have the correct number of dimensions or elements, this will result in a `NullPointerException` or an `ArrayIndexOutOfBoundsException`. If you use the value null (null pointer) as an input parameter (for `IN` and `INOUT` parameters), an array will be instantiated.

Mapping Groups and Periodic Groups

Groups (structures) in the IDL file are mapped to inner classes. If the Bean-compliant generation mode is used, they are mapped to normal classes in their own files. The group members (structure fields) are implemented as public fields of the inner class. If the bean-compliant generation is used, the members (structure fields) are implemented as private fields with getter and setter methods.

Example

The following example shows how to program with groups in a Java client and server. The IDL program consists of three groups, each with the same fields, but with different directions. The client shows how to initialize the fields in the groups for the In and InOut parameters and how to get the results from the Out and InOut parameters. The server part shows only the implemented server method, not the other parts of the generated server skeleton. The server just moves the data from the `In` parameters to the `Out` parameters and fills the gaps. We assume that `ClientGroup.class` and the client interface object Libgroup.class are in the same folder. To compile and run the client and the server you need the *entirex.jar*. For the server we assume that `LibgroupServer.class` and `LibgroupStub.class` are in the same folder and this folder is in the classpath of the EntireX RPC Server for Java.

IDL

```
library 'LibGroup' is
    program 'Program1' is
        define data parameter
        1 Group1    (/3)    In Out
            2 Field01   (A10)
            2 Field02   (N2)
            2 Field03   (I4)
        1 Group2    (/1)    In
            2 Field01   (A10)
            2 Field02   (N2)
            2 Field03   (I4)
        1 Group3    (/2)    Out
            2 Field01   (A10)
            2 Field02   (N2)
```

```
            2 Field03   (I4)
        end-define
```

Client

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientGroup {
   public static void main(String[] args) {
      try {
          Broker broker = new Broker(Libgroup.DEFAULT_BROKERID, "User1");
          broker.logon();
          // create the wrapper object.
          Libgroup lib = new Libgroup(broker, Libgroup.DEFAULT_SERVER);
          // /*
          //  * Using the old style:
          //  * Get the reference for group1 from wrapper object and
          //  * fill group1 with data. Since group1 is InOut, there exists a
          //  * reference.
          //  */
          // Group1[] group1 = lib.getGroup1();
          // for (int i = 0; i < group1.length; i++) {
          //    // create a new instance of each array element of group1.
          //    group1[i] = new Group1();
          //    // fill the data in each field.
          //    group1[i].setField01("group1 " + i);
          //    group1[i].setField02(new BigDecimal(i));
          //    group1[i].setField03(2 * i);
          // }
     /*
      * Fill the group1 parameters, using the new methods for indexed access.
      */
     Group1[] group1 = lib.getGroup1();
     for (int i = 0; i < group1.length; i++) {
       Group1 group = new Group1();
       group.setField01("group1 " + i);
       group.setField02(new BigDecimal(i));
       group.setField03(2 * i);
       lib.setGroup1(i, group);
     }

          /*
           *  Create an instance for group2. There is no reference for group2
           *  since this is an In parameter. Fill group2 with data.
           */
           Group1[] group2 = new Group1[1];
           for (int i = 0; i < group2.length; i++) {
              // create a new instance of each array element of group2.
              group2[i] = new Group1();
```

```
            // fill the data in each field.
            group2[i].setField01("group2 " + i);
            group2[i].setField02(new BigDecimal(i));
            group2[i].setField03(2 * i);
          }
          // do the RPC.
          lib.program1(group2);

          // /*
          //  * Using the old style:
          //  * We can use the reference group1, it is not modified.
          //  */
          // for (int i = 0; i < group1.length; i++) {
          //    // get the data from the group and print.
          //    System.out.println("Result of Program1; group1[" + i + "] "
          //      + group1[i].getField01() + ", " + group1[i].getField02() + ", "
          //       + group1[i].getField03());
          // }
      /*
       * Retrieve the group1 elements, using the new indexed access method.
       */
      for (int i = 0; i < 3; i++) {
        // get the data from the group and print.
        System.out.println("Result of Program1; group1[" + i + "] "
            + lib.getGroup1(i).getField01() + ", "
            + lib.getGroup1(i).getField02() + ", "
            + lib.getGroup1(i).getField03());
      }
          // /*
          //  * Using the old style:
          //  * Get the reference for group3. group3 is Out.
          //  */
          // Group1[] group3 = lib.getGroup3();
          // for (int i = 0; i < group3.length; i++) {
          //    // get the data from the group and print.
          //    System.out.println("Result of Program1; group3[" + i + "] "
          //      + group3[i].getField01() + ", " + group3[i].getField02() + ", "
          //       + group3[i].getField03());
          // }
      /*
       * Retrieve the group3 elements, using the new indexed access method.
       */
      for (int i = 0; i < 2; i++) {
        // get the data from the group and print.
        System.out.println("Result of Program1; group3[" + i + "] "
            + lib.getGroup3(i).getField01() + ", "
            + lib.getGroup3(i).getField02() + ", "
            + lib.getGroup3(i).getField03());
      }
          broker.logoff();
        } catch (BrokerException excep) {
          excep.printStackTrace ();
```

```
        }
    }

}
```

Client Group (Bean-compliant)

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientGroup {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libgroup.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libgroup lib = new Libgroup(broker, Libgroup.DEFAULT_SERVER);
            /* Get the reference for group1 from wrapper object and
             * fill group1 with data. Since group1 is InOut, there exists a
             * reference.
             */
            Group1[] group1 = lib.getGroup1();
            for (int i = 0; i < group1.length; i++) {
                // create a new instance of each array element of group1.
                group1[i] = new Group1();
                // fill the data in each field.
                group1[i].setField01("group1 " + i);
                group1[i].setField02(new BigDecimal(i));
                group1[i].setField03(2 * i);
            }
            /** Create an instance for group2. There is no reference for group2
             *  since this is an In parameter. Fill group2 with data.
             */
            Group1[] group2 = new Group1[1];
            for (int i = 0; i < group2.length; i++) {
                // create a new instance of each array element of group2.
                group2[i] = new Group1();
                // fill the data in each field.
                group2[i].setField01("group2 " + i);
                group2[i].setField02(new BigDecimal(i));
                group2[i].setField03(2 * i);
            }
            // do the RPC.
            lib.program1(group2);
            // We can use the reference group1, it is not modified.
            for (int i = 0; i < group1.length; i++) {
                // get the data from the group and print.
                System.out.println("Result of Program1; group1[" + i + "] "
                   + group1[i].getField01() + ", " + group1[i].getField02() + ", "
                    + group1[i].getField03());
```

```
            }
            // Get the reference for group3. group3 is Out.
            Group1[] group3 = lib.getGroup3();
            for (int i = 0; i < group3.length; i++) {
                // get the data from the group and print.
                System.out.println("Result of Program1; group3[" + i + "] "
                   + group3[i].getField01() + ", " + group3[i].getField02() + ", "
                    + group3[i].getField03());
            }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
    }

}
```

Server

```
public void program1 (LibgroupServer.Program1Group2[] group2) {
        /*
         * Program1Group1 is InOut
         * Program1Group2 is In
         * Program1Group3 is Out
         * Move the values from Program1Group2 to Program1Group1 and move the
         * value from Program1Group1 to Program1Group3.
         */
        int length = Math.min(program1Group1.length, program1Group3.length);
        for (int i = 0; i < length; i++) {
            if (program1Group3[i] == null)
                program1Group3[i] = new Program1Group3();
            program1Group3[i].field01 = program1Group1[i].field01;
            program1Group3[i].field02 = program1Group1[i].field02;
            program1Group3[i].field03 = program1Group1[i].field03;
        }
        for (int i = length; i < program1Group3.length; i++) {
            if (program1Group3[i] == null)
                program1Group3[i] = new Program1Group3();
            program1Group3[i].field01 = "New Text " + i;
            program1Group3[i].field02 = new BigDecimal(10);
            program1Group3[i].field03 = 100 + i;
        }
        // move the values from Program1Group1 to Program1Group3.
        length = Math.min(group2.length, program1Group1.length);
        for (int i = 0; i < length; i++) {
            if (program1Group1[i] == null)
                program1Group1[i] = new Program1Group1();
            program1Group1[i].field01 = group2[i].field01;
            program1Group1[i].field02 = group2[i].field02;
            program1Group1[i].field03 = group2[i].field03;
        }
```

```
        for (int i = length; i < program1Group1.length; i++) {
            if (program1Group1[i] == null)
                program1Group1[i] = new Program1Group1();
            program1Group1[i].field01 = "New Text " + i;
            program1Group1[i].field02 = new BigDecimal(10);
            program1Group1[i].field03 = 100 + i;
        }
    }
```

Mapping Structures

Structures are mapped like Groups. See *Mapping Groups and Periodic Groups*.

Example

The following example shows how to program with structures in a Java client and server. The structures are mapped to inner classes of the interface objects; if Bean-compliant generation is used, the structures are mapped to normal classes in their own file. The IDL program consists of one structure that is used with different directions. In the example above for the groups we have the same fields in each group. This example shows how to simplify this by using a structure. The structure is defined outside the program an references to the structure can be used several times in different programs. The client shows how to initialize the fields in the references of the structure for the In and InOut parameters and how to get the results from the Out and InOut parameters. The server part shows only the implemented server method, not the other parts of the generated server skeleton. The server just moves the data from the `In` parameters to the `Out` parameters and fills the gaps. We assume that `ClientStrct.class` and the client interface object `Libstrct.class` are in the same folder. To compile and run the client and the server you need the `entirex.jar`. For the server we assume that `LibstrctServer.class` and `LibstrctStub.class` are in the same folder and this folder is in the classpath of the EntireX RPC Server for Java.

IDL

```
library 'LibStrct' is
    struct 'Struct1' is
        define data parameter
        1 Field01   (A10)
        1 Field02   (N2)
        1 Field03   (I4)
        end-define

   program 'Program1' is
        define data parameter
        1 Ref1  ('Struct1'/3)   In Out
        1 Ref2  ('Struct1'/1)   In
        1 Ref3  ('Struct1'/2)   Out
        end-define
```

Client

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientStrct {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libstrct.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libstrct lib = new Libstrct(broker, Libstrct.DEFAULT_SERVER);
            /* create a struct object (as defined in the wrapper object) for the
             * InOut parameter struct1.
             */
            Struct1[] struct1 = new Struct1[3];
            // /*
            //  * Using the old style:
            //  *  fill the struct object with data.
            //  */
            // for (int i = 0; i < struct1.length; i++) {
            //     // create a new array element.
            //     struct1[i] = new Struct1();
            //     struct1[i].setField01("struct1 ");
            //     struct1[i].setField02(new BigDecimal(4 + i));
            //     struct1[i].setField03(i);
            // }
            // // set the struct object in the wrapper object
            // lib.setRef1 (struct1);
      /*
       * Fill the struct1 parameters, using the new methods for indexed access.
       */
            for (int i = 0; i < struct1.length; i++) {
              Struct1 struct = new Struct1();
                struct.setField01("struct1 ");
                struct.setField02(new BigDecimal(4 + i));
                struct.setField03(i);
                lib.setRef1(i, struct);
            }
            /* create a struct object (as defined in the wrapper object) for the
             * In parameter struct2.
             */
            Struct1[] struct2 = new Struct1[1];
            for (int i = 0; i < struct2.length; i++) {
                // create a new array element.
                struct2[i] = new Struct1();
                struct2[i].setField01("struct2 ");
                struct2[i].setField02(new BigDecimal(4 + i));
                struct2[i].setField03(i);
            }
```

```
            // do the RPC.
            lib.program1(struct2);

            // /*
            //  * Using the old style:
            //  * get the data from the InOut parameter struct1.
            //  */
            // for (int i = 0; i < struct1.length; i++) {
            //     // get the data from the struct and print.
            //     System.out.println("Result of Program1, struct1[" + i + "] "
           //        + struct1[i].getField01() + ", "   + struct1[i].getField02() ↩
+ ", "
            //         + struct1[i].getField03());
            // }
      /*
       * Retrieve the ref1 elements, using the new indexed access method.
       */
      for (int i = 0; i < 3; i++) {
        // get the data from the struct and print.
        System.out.println("Result of Program1, struct1[" + i + "] "
            + lib.getRef1(i).getField01() + ", "
            + lib.getRef1(i).getField02() + ", "
            + lib.getRef1(i).getField03());
      }
      // /*
      //  * Using the old style:
      //  * get the struct object for the Out parameter struct3.
      //  */
      // Struct1[] struct3 = lib.getRef3();
      // // get the data from the Out parameter struct3.
      // for (int i = 0; i < struct3.length; i++) {
      //     // get the data from the struct and print.
      //     System.out.println("Result of Program1, struct3[" + i + "] "
      //        + struct3[i].getField01() + ", "   + struct3[i].getField02() + ", "
      //         + struct3[i].getField03());
      // }
      /*
       * Retrieve the ref3 elements, using the new indexed access method.
       */
      for (int i = 0; i < 2; i++) {
        // get the data from the struct and print.
        System.out.println("Result of Program1, struct3[" + i + "] "
            + lib.getRef3(i).getField01() + ", "
            + lib.getRef3(i).getField02() + ", "
            + lib.getRef3(i).getField03());
      }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
```

```
    }
}
```

ClientStrct (Bean-compliant)

```
import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import java.math.BigDecimal;

public class ClientStrct {
    public static void main(String[] args) {
        try {
            Broker broker = new Broker(Libstrct.DEFAULT_BROKERID, "User1");
            broker.logon();
            // create the wrapper object.
            Libstrct lib = new Libstrct(broker, Libstrct.DEFAULT_SERVER);
            /* create a struct object (as defined in the wrapper object) for the
             * InOut parameter struct1.
             */
            Struct1[] struct1 = new Struct1[3];
            // fill the struct object with data.
            for (int i = 0; i < struct1.length; i++) {
                // create a new array element.
                struct1[i] = new Struct1();
                struct1[i].setField01("struct1 ");
                struct1[i].setField02(new BigDecimal(4 + i));
                struct1[i].setField03(i);
            }
            /* create a struct object (as defined in the wrapper object) for the
             * In parameter struct2.
             */
            Struct1[] struct2 = new Struct1[1];
            for (int i = 0; i < struct2.length; i++) {
                // create a new array element.
                struct2[i] = new Struct1();
                struct2[i].setField01("struct2 ");
                struct2[i].setField02(new BigDecimal(4 + i));
                struct2[i].setField03(i);
            }
            // set the struct object in the wrapper object
            lib.setRef1 (struct1);
            // do the RPC.
            lib.program1(struct2);
            // get the struct object for the Out parameter struct3.
            Struct1[] struct3 = lib.getRef3();
            // get the data from the InOut parameter struct1.
            for (int i = 0; i < struct1.length; i++) {
                // get the data from the struct and print.
                System.out.println("Result of Program1, struct1[" + i + "] "
                   + struct1[i].getField01() + ", "   + struct1[i].getField02() + ↩
", "
```

```
                    + struct1[i].getField03());
            }
            // get the data from the Out parameter struct3.
            for (int i = 0; i < struct3.length; i++) {
                // get the data from the struct and print.
                System.out.println("Result of Program1, struct3[" + i + "] "
                  + struct3[i].getField01() + ", "   + struct3[i].getField02() + ↩
", "
                    + struct3[i].getField03());
            }
            broker.logoff();
        } catch (BrokerException excep) {
            excep.printStackTrace ();
        }
    }
}
```

Server

```
public void program1 (Struct1[] ref2) {
        /*
         * Program1Group1 is InOut
         * Program1Group2 is In
         * Program1Group3 is Out
         * Move the values from Program1Group2 to Program1Group1 and move the
         * value from Program1Group1 to Program1Group3.
         */
        int length = Math.min(program1Ref1.length, program1Ref3.length);
        for (int i = 0; i < length; i++) {
            if (program1Ref3[i] == null)
                program1Ref3[i] = new Struct1();
            program1Ref3[i].field01 = program1Ref1[i].field01;
            program1Ref3[i].field02 = program1Ref1[i].field02;
            program1Ref3[i].field03 = program1Ref1[i].field03;
        }
        for (int i = length; i < program1Ref3.length; i++) {
            if (program1Ref3[i] == null)
                program1Ref3[i] = new Struct1();
            program1Ref3[i].field01 = "New Text " + i;
            program1Ref3[i].field02 = new BigDecimal(10);
            program1Ref3[i].field03 = 100 + i;
        }

       length = Math.min(ref2.length, program1Ref1.length);
        for (int i = 0; i < length; i++) {
            if (program1Ref1[i] == null)
                program1Ref1[i] = new Struct1();
            program1Ref1[i].field01 = ref2[i].field01;
            program1Ref1[i].field02 = ref2[i].field02;
            program1Ref1[i].field03 = ref2[i].field03;
        }
```

```
        for (int i = length; i < program1Ref1.length; i++) {
            if (program1Ref1[i] == null)
                program1Ref1[i] = new Struct1();
            program1Ref1[i].field01 = "New Text " + i;
            program1Ref1[i].field02 = new BigDecimal(10);
            program1Ref1[i].field03 = 100 + i;
        }
    }
```

Mapping the Direction Attributes In, Out, InOut

The IDL syntax allows you to define parameters as `IN` parameters, `OUT` parameters, or `IN OUT` parameters (which is the default if nothing is specified). This direction specification is reflected in the generated Java interface object as follows:

- `IN` parameters are sent from the RPC client to the RPC server. `IN` parameters are implemented as parameters of the generated method.
- `OUT` parameters are sent from the RPC server to the RPC client. `OUT` parameters are implemented as read-only properties. A `getMethod` is generated for each `OUT` parameter.
- `INOUT` parameters are sent from the RPC client to the RPC server and then back to the RPC client. `INOUT` parameters are implemented as properties. A `setMethod` and a corresponding `getMethod` is generated for each `INOUT` parameter.

Note that only the direction information of the top-level fields (level 1) is relevant. Group fields always inherit the specification from their parent. A different specification is ignored.

See the `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe attributes in the IDL file and refer to the `direction` attribute.

Mapping the aligned Attribute

The `aligned` attribute is not relevant for the programming language Java. However, a Java client can send the `aligned` attribute to an EntireX RPC server, where it might be needed.

See the `attribute-list` under *Software AG IDL Grammar* in the IDL Editor documentation for the syntax on how to describe attributes in the IDL file and refer to the `aligned` attribute.

Calling Servers as Procedures or Functions

The IDL syntax allows definition of procedures only. It does not have the concept of a function. A function is a procedure which, in addition to the parameters, returns a value. Procedures and functions are transparent between clients and servers. This means a client using a function can call a server implemented as a procedure and vice versa. In Java a procedure corresponds to a method with result type void, a function returns a value of some type.

It is possible to treat the `OUT` parameter of a procedure as the return value of a function. The Java Wrapper generates a method with a non-void result type when the following conditions are met:

- the last parameter of the procedure definition is of type `OUT`;
- this last parameter of the procedure definition has the name `Function_Result`. The name `Function_Result` is not case-sensitive.

Of course, in this case `getMethod` is not generated for this `OUT` parameter.

II Writing Applications with the Java Wrapper

5 Writing Simple Applications with the Java Wrapper

Required Steps

Interaction with the Java Wrapper occurs through instantiating objects of different classes, invoking their methods and manipulating their inner state. The basic steps for writing a client are listed below. Methods and properties to interact with the EntireX Broker are completely inherited from the EntireX Java ACI. The EntireX Java ACI also contains the class `RPCService` used as the superclass by the generated Java Wrapper classes.

Basic Steps:

- Instantiate a Broker object.

  One object instance represents one session to an EntireX Broker on your network. If you want to work with multiple EntireX Brokers or with multiple sessions, create one object for each session to an EntireX Broker.
- Use the Broker object to log the application on to EntireX Broker.
- Instantiate the generated Java Wrapper object (see *Java Wrapper Constructors*).
- Use the Java Wrapper methods (see *Generated Java Wrapper Methods*) to call the server programs and access their parameters.

Java Wrapper Constructors

Two constructors are available for the generated Java Wrapper class:

- public Example (Broker broker)
- public Example (Broker broker, String serverAddr)

public Example (Broker broker)

This constructor requires an instantiated Broker object only. The server address used is specified in the properties of the IDL file. Each generated Java Wrapper class has two public static String constants which contain the default values of the Broker and the server as set in the properties of the IDL file. For example:

```
public static final String DEFAULT_BROKERID = "localhost";
public static final String DEFAULT_SERVER = "RPC/SRV1/CALLNAT";
```

A Java Wrapper object using the default settings may be instantiated with the following coding:

```
Broker broker = new Broker(Example.DEFAULT_BROKERID, "UserId");
Example myExample = new Example(broker);
```

public Example (Broker broker, String serverAddr)

This constructor requires an instantiated Broker object and the server address. A Java Wrapper object can be instantiated with the following coding:

```
Broker broker = new Broker("localhost", "UserId");
Example myExample = new Example(broker, "RPC/MYRPC/CALLNAT");
```

Generated Java Wrapper Methods

EntireX Interface Object Version Information

To get the version information of the generated interface object, use the method `getStubVersion()`. It is implemented in the RPC client and server interface objects. The method returns a version string.

Example:

```
"EntireX RPC for Java Interface Object Version=10.5.0, Patch Level=0"
```

Application Identification

The application identification is sent from the application to the Broker. It is visible with Broker Command and Info Services. The identification consists of four parts: name, node, type, and version. These four parts are sent with each Broker call and are visible in trace information.

For the Java Wrapper these values are:

- **Application name**
  `ANAME=Java Runtime`
- **Node name**
  `ANODE=<host name>`

- **Application type**
  `ATYPE=Java`
- **Version**
  `AVERS=10.5.0.0`

The application is allowed to set the application name with the method `Broker.setApplicationName(String)`.

See `setApplicationName(java.lang.String)` of class `Broker` for more information.

6 Writing Advanced Applications - Java Wrapper

Each generated Java Wrapper class inherits methods from the EntireX Java ACI. This chapter describes what can be performed with the methods inherited from the class `RPCService`.

Logon to Natural Library

By default the library name sent to the RPC server is retrieved from the IDL file (see `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation). The library name can be overwritten. This is useful if communicating with a Natural RPC server.

To overwrite the library

1. Call the `setLibraryName` method of the generated Java Wrapper class with the new library name as a parameter.
2. Call the `setNaturalLogon` method of the generated Java Wrapper class with the parameter set to True.

If you need to set security credentials, refer to ***Using the Broker and RPC User ID/Password***.

Customizing the Generated Java Classes

You can extend the generated Java Wrapper Class of the client. By default, the generated client class is a subclass of `com.softwareag.entirex.aci.RPCService`. The customization component allows you to specify a class used as the superclass of the generated client class. This user-defined class (customization class) must be a subclass of `com.softwareag.entirex.aci.RPCService`.

When a customization class is specified, the calls to the user-exit methods `onEnter`, `onLeave`, `onException` and `onRetry` are generated.

To generate a customized Java Wrapper client

1. Implement your customization class. If you use a package for your customization class, specify package and class in the following step. Place the source for the customization class in the package folder, using the folder of the IDL file as package-root. The customization class needs a default constructor and one additional constructor with 4 arguments. See the example below.
2. Specify the name of your customization class in the *Designer*, under **Tools**, **Options**, **Java**. This name is stored in the *entirex.properties* file (which is in your home directory) using the key `entirex.wrapper.custom.class`.
3. Generate the wrapper client classes

To use the customized Java Wrapper client

- Add (public) arbitrary methods and fields to your customization class. These methods and fields are inherited by the generated client class. Add your own processing instructions to these methods.

To perform all Broker-related processing in the generated Java Wrapper client

1. Overwrite the constructors of `RPCService`. You can instantiate wrapper classes without specifying a Broker object and server address as a parameter. Use the method `setbroker()` to set or change the reference to the Broker object, and the method `setServerAddress()` to set or change the server address.
2. Use the four user exit methods `onEnter`, `onLeave`, `onException` and `onRetry`. These methods have default implementations in `RPCService` and can be overwritten in your customization class. These exits are called at the beginning and the end of each generated method of the Java Wrapper class and when a broker exception is thrown. See `RPCService`.

Example of a Customization Class

```
package ExamplePackage;

import com.softwareag.entirex.aci.Broker;
import com.softwareag.entirex.aci.BrokerException;
import com.softwareag.entirex.aci.RPCService;

public class ExampleCustomization extends RPCService {
   public ExampleCustomization ()
   {
      super();
   }
   public ExampleCustomization (Broker broker, String serverAddr, String
libName, boolean compress)
   {
      super(broker, serverAddr, libName, compress);
   }
   protected void onEnter(String progname) throws BrokerException {
      // insert your implementation here.
   }

   protected void onLeave(String progname, int sendLen, int receiveLen) throws
BrokerException {
      // insert your implementation here.
   }

   protected void onException(String progname, BrokerException exception) throws
BrokerException {
      // insert your implementation here.
   }
```

```
   protected boolean onRetry(String progname, BrokerException exception) throws
BrokerException {
      // insert your implementation here.
      return false;
   }
}
```

Using Conversational RPC

It is assumed that you are familiar with the concepts of conversational and non-conversational RPC. See *Conversational RPC*.

To enable conversational RPC

1 Create a Conversation object and set this with `setConversation` on the wrapper object.

2 Different wrapper objects can participate in the same conversation if they use the same instance of a conversation object.

To abort a conversational RPC communication

- Abort an RPC conversation by calling the `closeConversation` method

To close and commit a conversational RPC communication

- Commit the RPC conversation by calling the `closeConversationCommit` method.

Caution: Natural RPC Servers and EntireX RPC Servers behave differently when ending an RPC conversation.

See *Conversational RPC*.

Using the Broker and RPC User ID/Password

EntireX supports two user ID/password pairs: a broker user ID/password pair and an (optional) RPC user ID/password pair sent from RPC clients to RPC servers. With EntireX Security, the broker user ID/password pair can be checked for authentication and authorization.

The RPC user ID/password pair is designed to be used by the receiving RPC server. This component's configuration determines whether the pair is considered or not. Useful scenarios are:

- Credentials for Natural Security

- Impersonation in the respective RPC Server documentation
- Web Service Transport Security with the RPC Server for XML/SOAP, see *XML Mapping Files*
- Service execution with client credentials for EntireX Adapter Listeners, see *Configuring Listeners*
- etc.

Sending the RPC user ID/password pair needs to be explicitly enabled by the RPC client. If it is enabled but no RPC user ID/password pair is provided, the broker user ID/password pair is inherited to the RPC user ID/password pair.

With the `setNaturalLogon` (see below) sending the RPC user ID/password pair is enabled for the Java RPC clients. If you do so, we strongly recommend using SSL/TLS. See ***Using SSL/TLS***.

To use the broker and RPC user ID/password

1 Specify a broker user ID and optional broker password in the `logon` method of class `Broker`.

2 Call the `setNaturalLogon` method of the generated wrapper objects with the parameter set to `true` to enable sending the RPC user ID/password pair.

3 If different user IDs and/or passwords are used for broker and RPC, use the methods `setRPCUserId` and `setRPCPassword` to provide a different RPC user ID/password pair.

4 By default the library name sent to the RPC server is retrieved from the IDL file (see `library-definition` under *Software AG IDL Grammar* in the IDL Editor documentation). The library name can be overwritten. This is useful if communicating with a Natural RPC server. Specify a library in method `setLibraryName` of the generated wrapper objects.

If you need to log on to a Natural library without setting security credentials refer to ***Logon to Natural Library***.

Example:

Assume that library is a wrapper object that is generated from an IDL library. This object extends `com.softwareag.entirex.aci.RPCService`. For this object, call the methods as shown:

```
library.setRPCUserId("testuser");
library.setRPCPassword("password");
library.setLibraryName("NATLIB");  // this is necessary only if the Natural Library
                                   // name is different from the library name in ↩
the IDL.
library.setNaturalLogon(true);
```

The order of the four methods is arbitrary.

Using SSL/TLS

RPC client applications can use Secure Sockets Layer/Transport Layer Security (SSL/TLS) as the transport medium. The term “SSL” in this section refers to both SSL and TLS. RPC-based clients are always SSL clients. The SSL server can be either the EntireX Broker, Broker SSL Agent, or Direct RPC in webMethods Integration Server (IS inbound). For an introduction see *SSL/TLS and Certificates with EntireX* in the Platform-independent Administration documentation.

To use SSL

1 To operate with SSL, certificates need to be provided and maintained. Depending on the platform, Software AG provides default certificates, but we strongly recommend that you create your own. See *SSL/TLS Sample Certificates Delivered with EntireX* in the EntireX Security documentation.

2 Specify Broker ID and SSL parameters.

SSL transport will be chosen if the Broker ID starts with the string `ssl://`. Example of a typical *URL-style Broker ID*:

```
Broker broker = new Broker("ssl://yourbroker:10000?trust_store=castore","userID");
```

If no port number is specified, port 1958 is used as default.

If the SSL client checks the validity of the SSL server only, this is known as *one-way SSL*. The mandatory `trust_store` parameter specifies the file name of a keystore that must contain the list of trusted certificate authorities for the certificate of the SSL server. By default a check is made that the certificate of the SSL server is issued for the hostname specified in the Broker ID. The common name of the subject entry in the server's certificate is checked against the hostname. If they do not match, the connection will be refused. You can disable this check with SSL parameter `verify_server=no`.

If the SSL server additionally checks the identity of the SSL client, this is known as *two-way SSL*. In this case the SSL server requests a client certificate (the parameter `verify_client=yes` is defined in the configuration of the SSL server). Two additional SSL parameters must be specified on the SSL client side: `key_store` and `key_passwd`. This keystore must contain the private key of the SSL client. The password that protects the private key is specified with `key_passwd`.

The ampersand (&) character cannot appear in the password.

SSL parameters are separated by ampersand (&). See also *SSL/TLS Parameters for SSL Clients*.

Example of one-way SSL:

```
Broker broker = new ↩
Broker("ssl://yourbroker:10000?trust_store=castore&verify_server=no","userID");
```

Example of two-way SSL:

```
Broker broker = new ↩
Broker("ssl://yourbroker:10000?trust_store=castore&key_store=keystore&key_passwd=pwd","userID");
```

3 Make sure the SSL server to which the RPC component connects is prepared for SSL connections as well. The SSL server can be EntireX Broker, Broker SSL Agent, or Direct RPC in webMethods Integration Server (IS inbound). See:

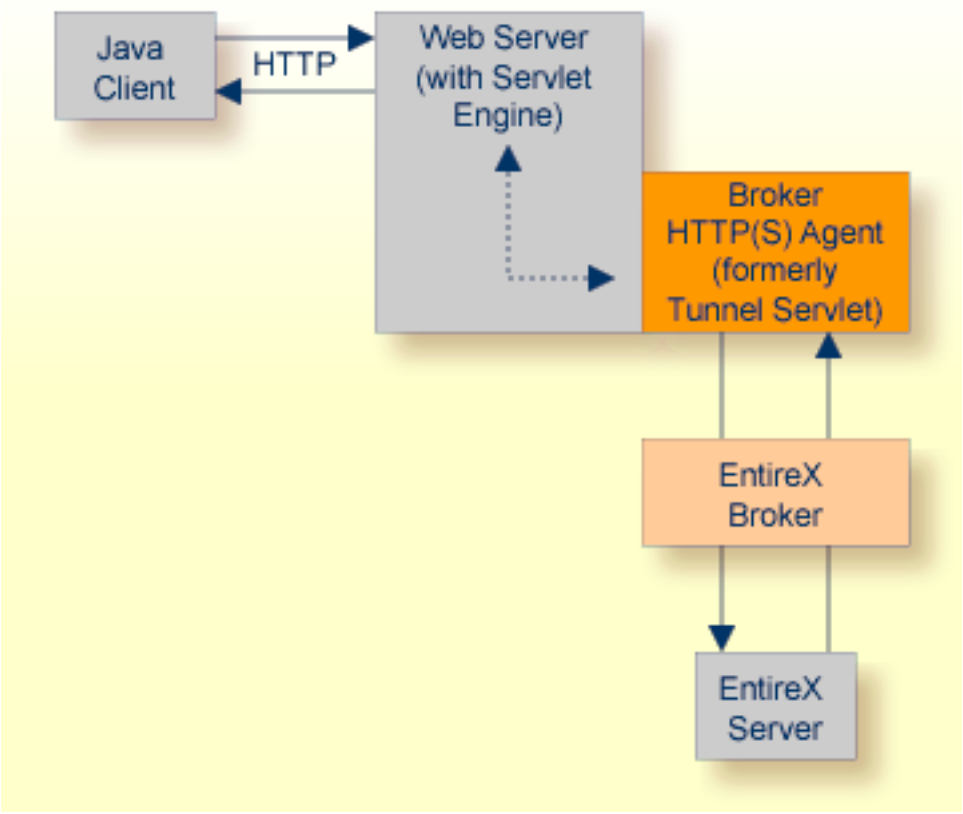
- *Running Broker with SSL/TLS Transport* in the platform-specific Administration documentation
- Broker SSL Agent in the UNIX and Windows Administration documentation
- *Support for SSL/TLS* in the EntireX Adapter documentation (for Direct RPC)

Using HTTP(S) Tunneling

When communicating with EntireX Broker over the internet, direct access to the EntireX Broker's TCP/IP port is necessary. This access is often restricted by proxy servers or firewalls. Java-based EntireX applications can pass communication data via HTTP or HTTPS. This means that a running EntireX Broker in the intranet is made accessible by a Web server without having to open additional TCP/IP ports on your firewall (HTTP tunneling). This section covers the following topics:

How the Communication Works

The Broker HTTP(S) Agent builds the bridge between Web server and EntireX Broker in the intranet.



The figure above shows how the communication works. In this scenario, a Java client program communicates via HTTP and EntireX Broker with an EntireX server. By using a Broker ID starting with "http://" (passing the URL of the installed Broker HTTP(S) Agent) each Broker request is sent to a Web server, which immediately processes the Broker HTTP(S) Agent, passes the contents to EntireX Broker, receives the answer and sends it back via HTTP. For the two partners (client and server) it is transparent that they are communicating through the Web. Java server programs can also communicate via HTTP if necessary.

For the configuration, see Broker HTTP(S) Agent in the UNIX and Windows Administration documentation.

Enabling HTTP Support

To enable HTTP support

- Pass the URL of your Broker HTTP(S) Agent installation as Broker ID to your Broker objects.

  For Example:

```
import com.softwareag.entirex.aci.Broker;

...
//  "http://www.yourhost.com/servlets/tunnel" is the URL to reach your broker ↩
over HTTP

Broker broker = new Broker("http://www.yourhost.com/servlets/tunnel","userID");
...

// other code not affected
...
```

The Broker HTTP(S) Agent optionally accepts parameters as part of the URL. It is possible to define values for Broker and log that override the corresponding values in the configuration of the Broker HTTP(S) Agent.

To enforce logging of the Broker HTTP(S) Agent

- Type, e.g. the following:

```
Broker broker = new ↩
Broker("http://www.yourhost.com/servlets/tunnel?log=yes","userID");
```

Enabling HTTPS Support

To use HTTPS instead of HTTP

- Replace "http://" by "https://" at the beginning of the Broker ID.

  Using HTTPS requires a Web server with SSL support enabled. Check your Web server's documentation for information on how to configure SSL support.

  Many Java implementations do not support HTTPS. If this is the case, your application will receive a BrokerException with error code 00130325.

Using Internationalization

RPC clients inherit all methods and functionality regarding character conversion, provided by the Java ACI. For more information see *Using Internationalization with Java ACI*.

Enable character conversion in the broker by setting the service-specific attribute `CONVERSION` to "`SAGTRPC`". See also *Configuring ICU Conversion* under *Configuring Broker for Internationalization* in the platform-specific Administration documentation. More information can be found under *Internationalization with EntireX*.

7 Writing RPC Clients for the RPC-ACI Bridge in Java

The EntireX RPC-ACI Bridge allows standard RPC clients to communicate with an ACI server. The RPC-ACI Bridge transforms RPC requests from clients into ACI messages.

The EntireX RPC-ACI Bridge reports errors from the RPC server side and the ACI side to the RPC clients. Errors from the ACI side include errors by the Broker for ACI. The RPC-ACI Bridge reports the same error classes and error codes for the RPC server side as the RPC Server for XML/SOAP. The RPC-ACI Bridge reports errors of the ACI side in a client-specific way as described below.

To write a Java client

1 Generate the Java RPC client stub from the IDL file as described in ***Using the Java Wrapper***.

2 Implement the client with this stub.

All errors are reported as `BrokerExceptions`. Errors on the ACI side of the RPC-ACI Bridge are `BrokerExceptions` in class 1018. See *Message Class 1018 - EntireX RPC-ACI Bridge*.

III Reliable RPC for Java Wrapper

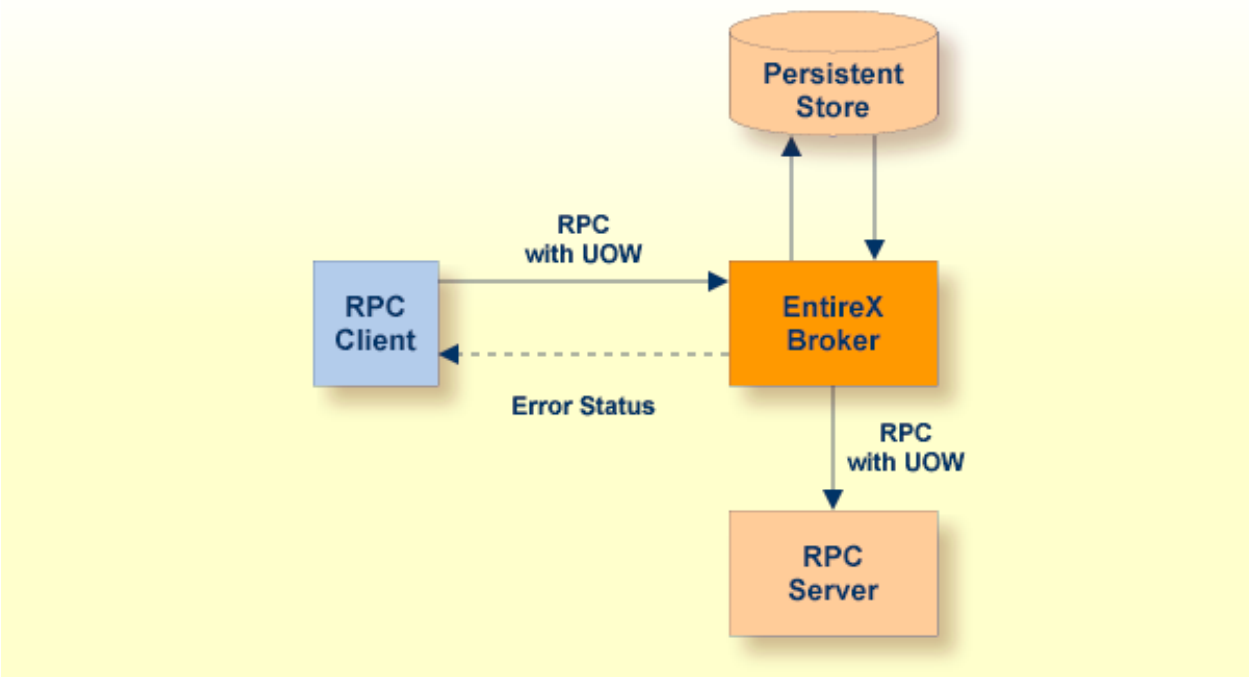
8 Reliable RPC for Java Wrapper

Introduction to Reliable RPC

In the architecture of modern e-business applications (such as SOA), loosely coupled systems are becoming more and more important. Reliable messaging is one important technology for this type of system.

Reliable RPC is the EntireX implementation of a reliable messaging system. It combines EntireX RPC technology and persistence, which is implemented with units of work (UOWs).

- Reliable RPC allows asynchronous calls (“fire and forget”)
- Reliable RPC is supported by most EntireX wrappers
- Reliable RPC messages are stored in the Broker's persistent store until a server is available
- Reliable RPC clients are able to request the status of the messages they have sent



Reliable RPC is used to send messages to a persisted Broker service. The messages are described by an IDL program that contains only `IN` parameters. The client interface object and the server interface object are generated from this IDL file, using the EntireX Java Wrapper.

Reliable RPC is enabled at runtime. The client has to set one of two different modes before issuing a reliable RPC request:

- `AUTO_COMMIT`
- `CLIENT_COMMIT`

While AUTO_COMMIT commits each RPC message implicitly after sending it, a series of RPC messages sent in a unit of work (UOW) can be committed or rolled back explicitly using CLIENT_COMMIT mode.

The server is implemented and configured in the same way as for normal RPC.

Writing a Client

All methods for reliable RPC are available on the interface object. See RPCService for details. The methods are:

- RPCService.setReliable
- RPCService.getReliable
- RPCService.reliableCommit
- RPCService.reliableRollback
- RPCService.getMessageId
- RPCService.getStatusOfMessage

Create Broker object and interface object.

```
Broker broker = new Broker(Mail.DEFAULT_BROKERID, userID);
Mail mail = new Mail(broker);
broker.logon();
```

Enable reliable RPC with CLIENT_COMMIT

```
  mail.setReliable(RPCService.RELIABLE_CLIENT_COMMIT);
```

The first RPC message.

```
  mail.sendmail("mail receiver", "Subject 1", "Text 1");
```

Check the status: get the message ID first and use it to retrieve the status.

```
String messageID = mail.getMessageID();
String messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The second RPC message.

```
mail.sendmail("mail receiver", "Subject 2", "Text 2");
```

Commit the two messages.

```
mail.reliableCommit();
```

Check the status again for the same message ID.

```
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

The third RPC message.

```
mail.sendmail("mail receiver", "Subject 3", "Text 3");
```

Check the status: get the new message ID and use it to retrieve the status.

```
messageID = mail.getMessageID();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
```

Roll back the third message and check status.

```
mail.reliableRollback();
messageStatus = mail.getStatusOfMessage(messageID);
System.out.println("Status: " + messageStatus + ", id: " + messageID);
broker.logoff();
```

Limitations

1. All program calls that are called in the same transaction (`CLIENT_COMMIT`) must be in the same IDL library.
2. It is not allowed to switch from `CLIENT_COMMIT` to `AUTO_COMMIT` in a transaction.
3. Messages (IDL programs) have `IN` parameters only.

Writing a Server

The server implementation consists of the four classes:

- `Abstract<`*`IDL library name`*`>Server`
- `<`*`IDL library name`*`>`
- `<`*`IDL library name`*`>Server`
- `<`*`IDL library name`*`>Stub`

Add your implementation to the class `<IDL library name>Server`. There are no server-side methods for reliable RPC. The server does not send back a message to the client. The server can run deferred, thus client and server do not necessarily run at the same time. If the server fails, it throws an exception. This causes a cancel of the transaction (unit of work inside the Broker) and the error code is written to the user status field of the unit of work.

Broker Configuration

A Broker configuration with `PSTORE` is recommended. This enables the Broker to store the messages for more than one Broker session. These messages are still available after Broker restart. The attributes `STORE`, `PSTORE`, and `PSTORE-TYPE` in the Broker attribute file can be used to configure this feature. The lifetime of the messages and the status information can be configured with the attributes `UOW-DATA-LIFETIME` and `UOW-STATUS-LIFETIME`. Other attributes such as `MAX-MESSAGES-IN-UOW`, `MAX-UOWS` and `MAX-UOW-MESSAGE-LENGTH` may be used in addition to configure the units of work. See *Broker Attributes*.

The result of the method `RPCService.getStatusOfMessage` depends on the configuration of the unit of work status lifetime. If the status is not stored longer than the message, the method returns `(not available)`.