Writing Advanced Applications - EntireX Java ACI

This chapter covers the following topics:

- Using Compression
- Using EntireX Security with Java-based EntireX Applications
- Using Integrated Authentication Framework with Java-based EntireX Applications
- Setting Transport Methods
- Tracing
- Using Internationalization with Java ACI

Using Compression

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) may compress the messages sent to and received from the Broker. There are two ways to enable compression:

- Use the method `setCompressionLevel()` of the Broker object.
- Use a Broker ID with the parameter `compresslevel=<value>`.

Using setCompressionLevel()

Add the compression level to the method `setCompressionLevel()` as an integer or a string argument.

You can use the constants defined in class `java.util.zip.Deflater`.

If the string

- starts with Y, compression is turned on with level 6,
- starts with N, compression is turned off (level 0).

Permitted values are the integers 0 - 9 and the corresponding strings:

BEST_COMPRESSION	level 9
BEST_SPEED	level 1
DEFAULT_COMPRESSION	level 6
DEFLATED	level 8
NO_COMPRESSION	level 0

Using Broker ID

You may append the keyword `COMPRESSLEVEL` with one of the values above to the Broker ID.

Examples

- ```
  localhost:1971?compresslevel=BEST_COMPRESSION
  ```
- ```
  localhost?poolsize=4&compresslevel=9
  ```

Both examples set the compression level to 9.

Using EntireX Security with Java-based EntireX Applications

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) which require security can use the security services offered by EntireX Security. See

- *Overview of EntireX Security* for a general introduction
-

Use the methods for security that are included in the Broker object (see `Broker`). The two security alternatives are

- using EntireX Security
- using your own security implementation.

To use EntireX Security

- Call one of the following methods for a Broker object:

  1. `useEntireXSecurity()`
  2. `useEntireXSecurity(int encryptionLevel)`
  3. `useEntireXSecurity(boolean autoMode)`
  4. `useEntireXSecurity(int encryptionLevel, boolean autoMode)`

  You can set the encryption level with this call and you can enable the auto mode. The encryption level allows the values ENCRYPTION_LEVEL_NONE, where the message is not encrypted, ENCRYPTION_LEVEL_BROKER, where the message is encrypted on the way to the EntireX Broker, and ENCRYPTION_LEVEL_TARGET, where the message is encrypted the whole way to the target. The auto mode specifies that the Broker object uses the EntireX Security as needed by the EntireX Broker. If the EntireX Broker uses security, the EntireX Security object is used by the Broker object.

The method `useEntireXSecurity` must be called before the first call of `logon`, which must use a password. The security object cannot change during a session with the EntireX Broker.

To use your own security implementation

- Implement the interface BrokerSecurity. This implementation must have an accompanying security exit for the EntireX Broker. See Using Sample Security Exits for Broker Security. Call the methods `setSecurity` with the security object and set encryption level or auto mode in the same way as the `useEntireXSecurity` methods.

  An example of EntireX Security can be found in the *Client.java* source in the Java ACI examples. Set the constant field `SECURITY` to true, support a password to the logon method and compile the source.

Using Integrated Authentication Framework with Java-based EntireX Applications

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) which require security can use the security services offered by IAF.

The methods for IAF are included in the `Broker` object (see `Broker`). If the Broker is set up for IAF, the client application can get the IAF token after logon with the method `Broker.getIAFToken`. The token is a byte[256] array. The content is not visible to the client. The client can use the token as-is only and must not change it. The token returned by `Broker.getIAFToken` can be used to authenticate the client to other products using IAF.

On the other hand, a token obtained from some other product can be used with `Broker.setIAFToken` to authenticate with the Broker.

The client should delete the token after `Broker.logoff` with `Broker.setIAFToken(null)`.

Setting Transport Methods

- Socket Parameters for TCP and SSL Communication
- Using SSL
- Using HTTP(S) Tunneling
- Setting the Transport Timeout

Socket Parameters for TCP and SSL Communication

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) up to version 5.3.1 use one socket connection for each instance of the Broker class.

Starting with EntireX version 6.1.1, a pool of socket connections is managed by the EntireX Java runtime.

Socket connections are

- assigned dynamically to instances of Broker objects
- closed automatically when they are not used for a certain period of time.

Controlling Socket Pooling

The behavior of the socket pooling can be controlled by two parameters (`poolsize` and `pooltimeout`) specified as part of the Broker ID. They are used for both TCP and SSL communications.

You can

- specify the maximum number of socket connections which are kept in the socket pool
- disable socket pooling
- control the automatic closing of socket connections

To specify the maximum number of socket connections

- Specify the parameter `poolsize` as part of the Broker ID.

  If the number entered is reached, further Broker calls going through a Broker instance will be delayed until a socket becomes available. If a multithreaded application uses blocking sendReceive or Receive calls with a longer waiting time, the `poolsize` parameter must be at least equal to the number of threads. Starting with EntireX version 7.1.1.60, the value of `entirex.timeout` (in seconds) is used to terminate the wait time for free sockets. If all sockets in the pool are in use, the calls will be delayed at the most by the period of time specified by this timeout. Afterwards, the call returns with error code 0013 0333. This is to prevent applications from hanging up if all sockets are in use and never become available due to network problems.

  The default for `poolsize` is 32. The default can be changed with a Java system property. Set the property `entirex.socket.poolsize` to specify a different value. Values that are not numeric or less than 1 are ignored.

To disable socket pooling

- Set the parameter `poolsize` (as part of the Broker ID) to "0".

  The behavior is then identical to that of the pre-6.1.1 versions of EntireX.

To control the automatic closing of socket connections

- Specify the parameter `pooltimeout` (as part of the Broker ID).

  If a socket connection has not been used for the specified number of seconds, it will be closed automatically.

  The default for `pooltimeout` is 300 seconds.

  Example of a maximum number of 10 socket connections and a timeout of 60 seconds:

```
Broker broker = new Broker("yourbroker?poolsize=10&pooltimeout=60","userID");
```

Using SSL

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) can use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) as the transport medium. In this section, "SSL" refers to both SSL and TLS. Java-based clients or servers are always SSL clients. The SSL server can be either the EntireX Broker or the EntireX Broker Agent. SSL transport will be chosen if the Broker ID starts with the string ssl://.

Example of a typical Broker ID for SSL:

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

If no port number is specified, port 1958 will be used as the default port. The `trust_store` parameter is mandatory. It specifies the file name of a Java keystore that must contain the list of trusted certificate authorities for the certificate of the SSL server. If the server requests a client certificate (the parameter `verify_client=yes` is defined in the configuration of the SSL server) two additional parameters have to be specified as part of the Broker ID:

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

Again, *key_store* is the file name of a Java keystore. 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.

By default a check is made that the certificate of the SSL server is issued for the hostname specified in the Broker ID. In the example above, the common name of the subject entry in the server's certificate must be identical to `yourbroker`. This checking can be disabled by specifying the parameter `verify_server=no` in the Broker ID.

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 (including applications using classes generated by the *Java Wrapper*) 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).

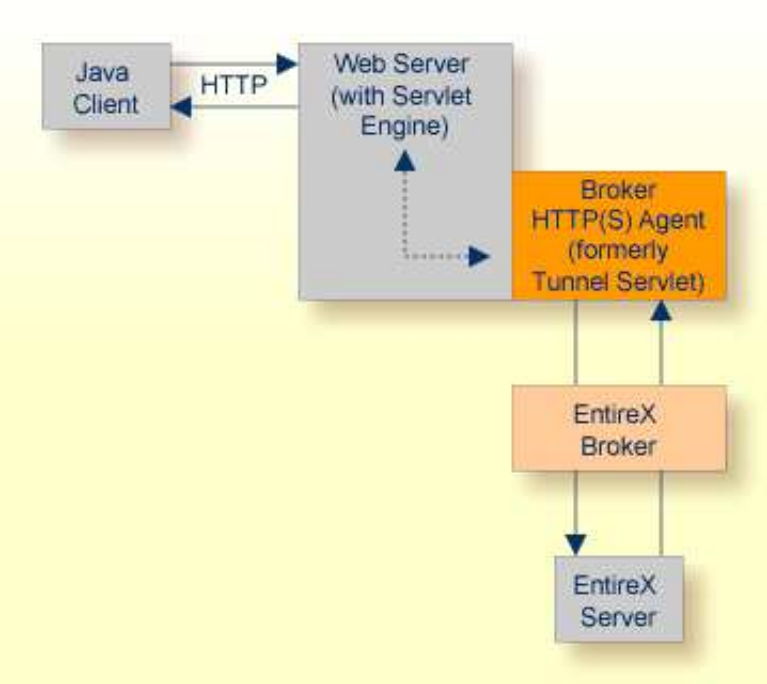
This section covers the following topics:

- How the Communication Works
- How to Enable HTTP Support in a Java Component
- How to Enable HTTPS Support in a Java Component

How the Communication Works

The EntireX Java ACI is able to send and receive data via an HTTP protocol controlled by constructor `com.softwareag.entirex.aci.Broker`. See *How to Enable HTTP Support in a Java Component*.

The EntireX Java component `com.softwareag.entirex.aci.TunnelServlet.class` implements a Java servlet for servlet-enabled Web servers. It 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 HTTP(S) Agent (formerly referred to as Tunnel Servlet)) each Broker request is sent to a Web server, which immediately processes the 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 *Settting up and Administering the Broker HTTP(S) Agent* under UNIX | Windows.

How to Enable HTTP Support in a Java Component

To enable HTTP support for a Java-based component

- Pass the URL of your 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 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 HTTP(S) Agent.

To enforce logging of the HTTP(S) Agent

- Type, e.g. the following:

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

How to Enable HTTPS Support in a Java Component

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.

Setting the Transport Timeout

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) can set a transport timeout to abort socket connections when not receiving any reply.

To specify a TCP or SSL transport timeout

1. Use the system property `entirex.timeout`.

   The default is 20 seconds.

   A numeric value of 1 or greater indicates the transport timeout in seconds.

   Setting the value to 0 results in a potentially infinite wait (i.e. until the Broker returns a reply or the socket connection is closed).

   If the Broker call is a send call with wait or a receive call, the transport timeout is added to the Broker wait time specified as part of the Broker call.

   The value of `entirex.timeout` is used as a timeout for waiting for free sockets in the socket pools. If the application does not get a free socket during this timeout period, an exception will be thrown.

2. Use the static method `Broker.setTransportTimeout(int timeout)` in your application.

   This method sets the socket timeout value in seconds. It is used for TCP/IP, but not with HTTP. The timeout value is used for new sockets, it does not change the timeout for sockets in use.

   To query the current setting, use the method `Broker.getTransportTimeout()`.

Tracing

Java-based EntireX applications (including applications using classes generated by the *Java Wrapper*) can use tracing to log program flow and locate problems.

To specify the trace level

- Use the `setTrace()` method of class `Broker`.

  Or:
  Use the Java system property `entirex.trace`. The system property uses the same values as the `setTrace` method call.

Trace level	Explanation
0	no tracing, default.
1	trace all Broker calls and other major actions
2	dump the send and receive buffer
3	dump the buffers sent to the Broker and received from the Broker

Using Internationalization with Java ACI

It is assumed that you have read the document *Internationalization with EntireX* and are familiar with the various internationalization approaches described there.

EntireX Java components use the codepage configured for the Java virtual machine (JVM) to convert the Unicode (UTF-16) representation within Java to the multibyte or single-byte encoding sent to or received from the broker by default. This codepage is also transferred as part of the locale string to tell the broker the encoding of the data if communicating with a broker version 7.2.x and above.

To change the default, see your JVM documentation. On some JVM implementations, it can be changed with the `file.encoding` property. On some UNIX implementations, it can be changed with the `LANG` environment variable.

Which encodings are valid depends on the version of your JVM. For a list of valid encodings, see *Supported Encodings* in your Java documentation. The encoding must also be a supported codepage of the broker, depending on the internationalization approach.

With the `setCharacterEncoding(enc)` method of the `BrokerService` (EntireX Java ACI) you can

- override the encoding used for the payload sent to / received from the broker. Instead of using the default JVM encoding, the given encoding is used. Using this method does not change the default encoding of your JVM.
- force a locale string to be sent if communicating with broker version 7.1.x and below. Use the value LOCAL to send the default encoding of the JVM to the broker. See *Using the Abstract Codepage Name LOCAL*.