CAPLIN

Caplin Xaqua 1.0

How To Create A Permissioning DataSource Adapter

November 2011

Contents

1 Preface

1.1 What this document contains

This document describes how you can create a Permissioning DataSource adapter by writing an application that uses the Permissioning DataSource API. A Permissioning DataSource adapter is required to integrate Caplin Xaqua with a Permissioning System. The document also discusses the Demo Permissioning DataSource that is provided with the reference implementation of Caplin Trader from release 1.2.8.

Before reading this document, make sure you are familiar with the document **Caplin Xaqua: Permissioning Overview And Concepts**.

About Caplin document formats

This document is supplied in three formats:

- Portable document format (*.PDF* file), which you can read on-line using a suitable PDF reader such as Adobe Reader®. This version of the document is formatted as a printable manual; you can print it from the PDF reader.
- Web pages (*.HTML* files), which you can read on-line using a web browser. To read the web version of the document navigate to the *HTMLDoc_m_n* folder and open the file *index.html*.
- Microsoft HTML Help (*.CHM* file), which is an HTML format contained in a single file.
  To read a *.CHM* file just open it – no web browser is needed.

For the best reading experience

On the machine where your browser or PDF reader runs, install the following Microsoft Windows® fonts: Arial, Courier New, Times New Roman, Tahoma. You must have a suitable Microsoft license to use these fonts.

Restrictions on viewing .CHM files

You can only read *.CHM* files from Microsoft Windows.

Microsoft Windows security restrictions may prevent you from viewing the content of *.CHM* files that are located on network drives. To fix this either copy the file to a local hard drive on your PC (for example the Desktop), or ask your System Administrator to grant access to the file across the network. For more information see the Microsoft knowledge base article at http://support.microsoft.com/kb/896054/.

1.2 Who should read this document

This document is intended for System Architects and Software Developers who want to integrate Caplin Xaqua with a Permissioning System.

1.3 Related documents

- **Caplin Xaqua: Overview**

  Provides a business and technical overview of Caplin Xaqua and includes an explanation of its architecture.

- **Caplin Liberator: Administration Guide**

  Describes how to install and configure Caplin Liberator and discusses the authentication modules that are provided with the server.

- **Caplin Xaqua: Permissioning Overview And Concepts**

  Introduces permissioning concepts and terms, and shows the permissioning components of the Caplin Xaqua architecture.

- **Caplin Xaqua: Installing Permissioning Components**

  Describes how to install the Permissioning Auth Module and Permissioning DataSource in an existing Caplin Xaqua installation. You only need to install these components if your installation of Caplin Trader is earlier than release 1.2.8, as later releases include these permissioning components.

- **Caplin Trader: How To Add Permissioning At The Client**

  Describes how to add Permissioning to Caplin Trader.

- **Permissioning DataSource: API Reference**

  The API reference documentation provided with the Permissioning DataSource SDK (Software Development Kit). The classes and interfaces presented by this API allow you to write a Java application that will integrate a Permissioning System with Caplin Xaqua.

- **Caplin Trader: API Reference**

  The API reference documentation provided with Caplin Trader. The classes and interfaces of the `caplin.security.permissioning` package allow you to write JavaScript classes that extend the live permissioning capabilities of Caplin Trader.

1.4 Typographical conventions

The following typographical conventions are used to identify particular elements within the text.

Type	***Uses***
aMethod	Function or method name
aParameter	Parameter or variable name
/AFolder/Afile.txt	File names, folders and directories
`Some code;`	Program output and code examples
The `value=10` attribute is...	Code fragment in line with normal text
Some text in a dialog box	Dialog box output
`Something typed in`	User input – things you type at the computer keyboard
Glossary term	Items that appear in the “Glossary of terms and acronyms”
XYZ Product Overview	Document name
◆	Information bullet point
■	Action bullet point – an action you should perform

Note: Important Notes are enclosed within a box like this.
Please pay particular attention to these points to ensure proper configuration and operation of the solution.

Tip: Useful information is enclosed within a box like this.
Use these points to find out where to get more help on a topic.

Information about the applicability of a section is enclosed in a box like this.
For example: “This section only applies to version 1.3 of the product.”

1.5 Feedback

Customer feedback can only improve the quality of our product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Visit our feedback web page at https://support.caplin.com/documentfeedback/.

1.6 Acknowledgments

Adobe® Reader is a registered trademarks and *Adobe Flex™* a trademark of Adobe Systems Incorporated in the United States and/or other countries.

Windows is a registered trademark and *Silverlight™* a trademark of Microsoft Corporation in the United States and other countries.

Java, JavaScript, and *JVM* are trademarks or registered trademarks of Oracle® Corporation in the U.S. and other countries.

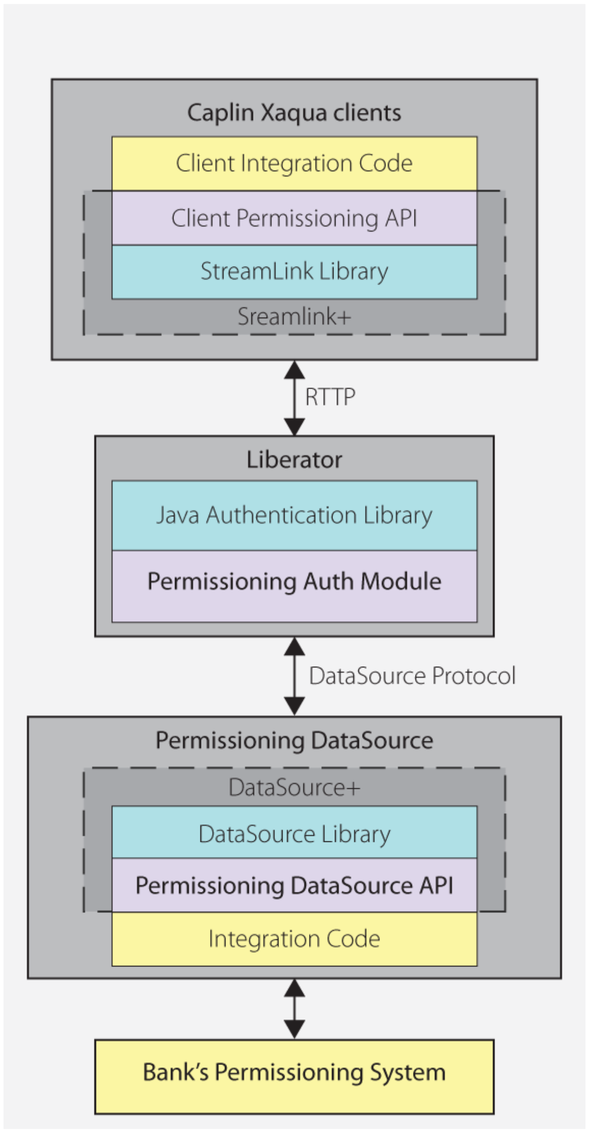
1.7 Code samples in this document

The code samples presented in this document use the following conventions:

- Text within `<angled brackets>` represents parameters that must be defined in your code.
- Text shown as `( ... )` represents parameters that have been omitted for simplicity.

2 What is a Permissioning DataSource?

A **Permissioning DataSource** is a **DataSource Adapter** that acts as the interface between **Caplin Xaqua** and your **Permissioning System**. Its purpose is to provide **Liberator** with the permissioning data that the **Permissioning Auth Module** will use to decide whether or not an interaction with Liberator is permitted.



Simplified Caplin Xaqua architecture showing only permissioning components

To create a Permissioning DataSource, you write and compile a Java application that uses the **Permissioning DataSource API**. This simple **API** is built on top of the Caplin **DataSource** for Java API, allowing your application to send permissioning data to Liberator using the **DataSource protocol**, but without the need for your code to explicitly use the DataSource API.

Tip: You will find further information about the permissioning components of the Caplin Xaqua architecture in the document **Caplin Xaqua: Permissioning Overview And Concepts**.

2.1 The Permissioning DataSource API

The Permissioning DataSource API is part of the Permissioning DataSource SDK (Software Development Kit) and allows you to write applications that can send permissioning data to Caplin Liberator. The SDK is delivered with Caplin Xaqua and contains the following components.

- The library of Java classes that provide the Permissioning DataSource API.
- **Permissioning DataSource: API Reference** that includes an overview, and package and class-level documentation.
- A Demo Permissioning DataSource Adapter 43. This example application uses the Permissioning DataSource API to provide Liberator with permissioning data from an XML file 46.

The Permissioning DataSource API is contained in a single package that provides the classes and interfaces you need to integrate Caplin Xaqua with a Permissioning System. The package also includes classes for assigning permissions to **Users** and **Groups**, classes for storing permissioning data, and classes for handling exceptions.

Tip: For a complete description of the Permissioning DataSource API, please refer to the **Permissioning DataSource: API Reference**.

3 Creating a Permissioning DataSource Adapter

Permissioning data can either be sent to Liberator from a single Permissioning DataSource 7, or from multiple Permissioning DataSources 11.

Note: To create a Permissioning DataSource adapter as described in this document, your application must use version 4.5.18 (or later) of the Permissioning DataSource library. Please contact Caplin support if you intend to use an earlier version of that library.

3.1 Creating a Single Permissioning DataSource

The Permissioning DataSource API provides the interface between Caplin Xaqua and a Permissioning System. When you write an application that uses this API, your code must implement the `PermissioningConnectionListener` interface and instantiate a `PermissioningDataSource`, as summarized in steps 1 to 5 below.

1. Implement the `PermissioningConnectionListener` interface

   This interface has two callback methods that your code must implement. The first of these callback methods, `onConnect()`, is triggered by the `PermissioningDataSource` when a new connection to Liberator is established. The second of these callback methods, `onDisconnect()`, is triggered by the `PermissioningDataSource` when the connection to Liberator is lost.

   These are informational callbacks only, and an application would typically respond by logging the current connection status.

2. Instantiate a `PermissioningDataSource`

   The `PermissioningDataSource` has one constructor that expects three arguments in the following order:

   - An instance of your `PermissioningConnectionListener` implementation, as described in step 1 above.
   - A DataSource XML configuration file (*conf/DataSource.xml*), in the form of an `InputStream`. This file configures the `PermissioningDataSource` as a DataSource adapter, and must contain network connection information for your particular network.
   - A DataSource XML field mapping file (*conf/Fields.xml*), in the form of an `InputStream`. This file maps DataSource field names to field numbers, and must match the field name to number mappings that are used by Liberator.

   The Demo Permissioning DataSource 43 that is supplied with the **SDK** has an example DataSource XML configuration file and example DataSource XML field mapping file. You can either create your own version of these files or customize the supplied example files as required.

3. Set the **role** of the `PermissioningDataSource`

   When there is only one `PermissioningDataSource` connected to Liberator, set the role to **master** (see Creating Multiple Permissioning DataSource Adapters 11).

Note: If your client application does not support multiple Permissioning DataSources, then omit step 3 and do not set the role of the `PermissioningDataSource` (see Upgrading the Permissioning DataSource library 9).

4. Apply permissioning data to the `PermissioningDataSource`

   Permissioning **rules**, and the permissions of **users** and **groups**, are created as part of a transaction and applied to the `PermissioningDataSource` when the transaction is committed. Permissioning data can be applied to the `PermissioningDataSource` at any time, either before or after a Liberator connection is established.

   Each time a connection is established, the `PermissioningDataSource` sends permissioning data for all committed transactions to Liberator as an image. If a connection to Liberator is already established, permissioning data is sent to Liberator as soon as the transaction is committed.

5. Start the `PermissioningDataSource`

   You start a `PermissioningDataSource` when you call `PermissioningDataSource.start().`

The following code sample is a trivial implementation of a `PermissioningConnectionListener`, as summarized in steps 1 to 5 above.

```
// Step 1 (part 1/3): Implement the PermissioningConnectionListener interface.
public class MyPermissioningSystemAdapter implements PermissioningConnectionListener
{
   private PermissioningDataSource pds;

   public MyPermissioningSystemAdapter() throws IOException, SAXException
   {
      // Step 2: Instantiate a PermissioningDataSource, passing in this
      // adapter as a listener.
      pds = new PermissioningDataSource(this,
                                        <DataSource.Config.Stream>,
                                        <Fields.Config.Stream>);

      // Step 3: Set the role of the PermissioningDataSource to master.
      pds.setMasterRole();

      // Step 4: In this example we apply permission data before starting
      // the PermissioningDataSource, but this can be done at any time.
      // Apply the permissioning data as part of an image transaction.
      pds.startImageTransaction();

      // Create some Rules.
      pds.createActionRule( ... );
      pds.createActionRefRule( ... );

      // Create some Users and configure them.
      User user1 = pds.createUser( ... );
      user1.applyPermission( ... );
      user1.setSubjectMapping( ... );

      User user2 = pds.createUser( ... );
      user2.applyPermission( ... );

      // Create some Groups and configure them.
      Group group1 = pds.createGroup( ... );
      group1.applyPermission( ... );
      group1.addMember( user1 );

      Group group2 = pds.createGroup( ... );
      group2.applyPermission( ... );
      group2.addMember( user1 );
      group2.addMember( user2 );

      // Commit the permissioning data. The data is sent to Liberator immediately,
      // or as soon as a connection to Liberator is established.
      pds.commitTransaction();

      // Step 5: Start the PermissioningDataSource.
      pds.start();
   }
   // Step 1 (part 2/3): Implement the onConnect() callback
   public void onConnect(int peerIndex)
   {
      System.out.println("Connected to Liberator!")
   }
   // Step 1 (part 3/3): Implement the onDisconnect() callback
   public void onDisconnect(int peerIndex)
   {
      System.out.println("Disconnected from Liberator!")
   }
}
```

Upgrading the Permissioning DataSource library

From release 4.5.6, the Permissioning DataSource library supports two versions of a Permissioning message protocol, each having a different (and mutually incompatible) message format.

Version 1 (the original protocol) has a message format that allows only one Permissioning DataSource to connect to Liberator. Version 2 (a later protocol) has a message format that allows both single (**master**) and multiple (master/**slave**) Permissioning DataSources to connect to Liberator.

If you are you are upgrading the Permissioning DataSource library and your client application uses version 1 of the Permissioning message protocol, then you must ensure that your Permissioning DataSource continues to use version 1 of this protocol.

A Permissioning DataSource will use version 1 of the protocol if you *do not* set the role of the Permissioning DataSource (see step 3 of Creating a Single Permissioning DataSource 7). This means that if the client application only supports version 1 of the protocol, then you *do not* need to modify any code in either the client application or Permissioning DataSource when you upgrade the Permissioning DataSource library.

A Permissioning DataSource will use version 2 of the protocol if you *do* set the role of the Permissioning DataSource. You must set the role of the Permissioning DataSource if your client application is configured to use version 2 of the Permissioning message protocol.

The following table shows the messaging protocols that are supported by each release of the Permissioning DataSource and **Caplin Trader** libraries.

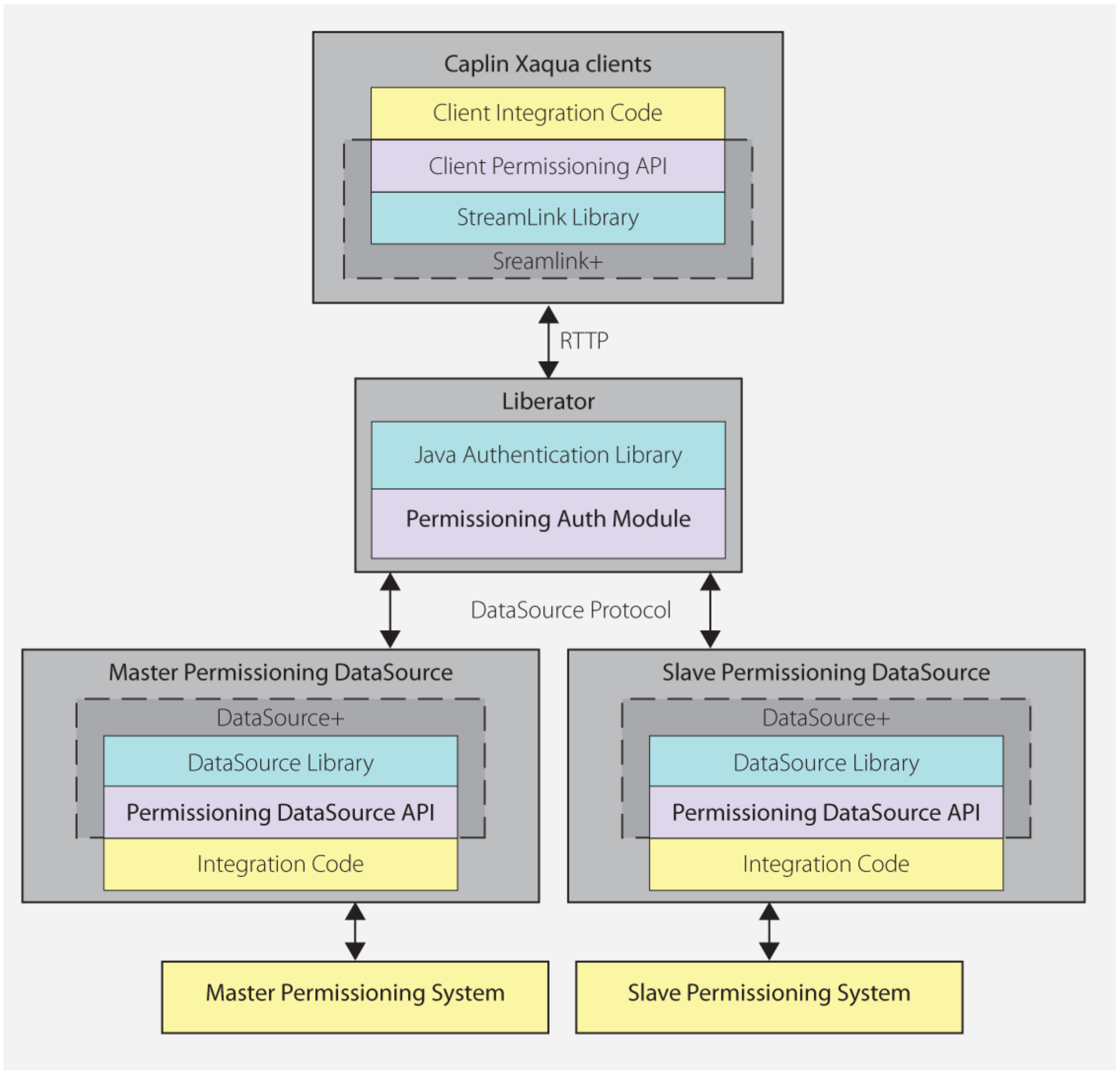
Supported Permissioning message protocols:

Component	Release	Supported Permissioning Message Protocol	How to configure
Permissioning DataSource library (DataSource+)	4.5.3 and earlier	version 1 only	Not applicable
Permissioning DataSource library (DataSource+)	4.5.4 and 4.5.5	version 2 only	Not applicable

Component	**Release**	**Supported Permissioning Message Protocol**	**How to configure**
Permissioning DataSource library (DataSource+)	4.5.6 and later	versions 1 and 2	The Permissioning DataSource will use protocol version 1 if you *do not* set the master or slave role. The Permissioning DataSource will use protocol version 2 if you *do* set the master or slave role.
Caplin Trader library (StreamLink+)	1.4.8 and earlier	version 1 only	Not applicable
Caplin Trader library (StreamLink+)	1.5.0 and later	version 1 and 2	See the document **Caplin Trader: How To Add Permissioning At The Client**.

3.2 Creating Multiple Permissioning DataSource Adapters

When permissioning data is sent to Liberator from more than one Permissioning DataSource, one of the Permissioning DataSources must be designated the master and each of the other Permissioning DataSources must be designated as slaves.


Caplin Xaqua clients
Client Integration Code
Client Permissioning API
StreamLink Library
Sreamlink+
RTTP
Liberator
Java Authentication Library
Permissioning Auth Module
DataSource Protocol
Master Permissioning DataSource
DataSource+
DataSource Library
Permissioning DataSource API
Integration Code
Slave Permissioning DataSource
DataSource+
DataSource Library
Permissioning DataSource API
Integration Code
Master Permissioning System
Slave Permissioning System

Multiple Permissioning DataSource Adapters connected to Liberator (showing one master and one slave)

There can only be one master Permissioning DataSource, but there can be one or more slave Permissioning DataSources depending on business requirements. For example, one slave could provide permissions for FX instruments and another permissions for FI instruments. Only the master can add permissioning rules and the user authentication details that allow end-users to log in to Liberator (see Master/Slave Limitations 14).

Creating the Master

To designate a Permissioning DataSource as the master, your code must call methods that set the role of the Permissioning DataSource as master. In the following code sample, the master is set in step 3. The code is identical to the code sample described in Creating a Permissioning DataSource Adapter 7, except that user permissions and subject mappings are set in the slave (see Creating a Slave 13).

```
// Step 1 (part 1/3): Implement the PermissioningConnectionListener interface.
public class MasterPermissioningSystemAdapter implements PermissioningConnectionListener
{
   private PermissioningDataSource pdsm;

   public MasterPermissioningSystemAdapter() throws IOException, SAXException
   {
      // Step 2: Instantiate a PermissioningDataSource, passing in this
      // adapter as a listener.
      pdsm = new PermissioningDataSource(this,
                                  <DataSource.Config.Stream>,
                                  <Fields.Config.Stream>);

      // Step 3: Set the role of the PermissioningDataSource to master.
      pdsm.setMasterRole();

      // Step 4: In this example we apply permission data before starting
      // the PermissioningDataSource, but this can be done at any time.
      // Apply the permissioning data as part of an image transaction.
      pdsm.startImageTransaction();

      // Create some Rules.
      pdsm.createActionRule( ... );
      pdsm.createActionRefRule( ... );

      // Create some Users - permissions and subject mappings for these
      // Users are applied in the slave, but could be applied here.
      User user1 = pdsm.createUser( ... );
      User user2 = pdsm.createUser( ... );

      // Create some Groups and configure them.
      Group group1 = pdsm.createGroup( ... );
      group1.applyPermission( ... );
      group1.addMember( user1 );

      Group group2 = pdsm.createGroup( ... );
      group2.applyPermission( ... );
      group2.addMember( user1 );
      group2.addMember( user2 );

      // Commit the permissioning data. The data is sent to Liberator immediately,
      // or as soon as a connection to Liberator is established.
      pdsm.commitTransaction();

      // Step 5: Start the PermissioningDataSource.
      pdsm.start();
   }

   // Step 1 (part 2/3): Implement the onConnect() callback
   ...

   // Step 1 (part 3/3): Implement the onDisconnect() callback.
   ...
}
```

Note that permissions and subject mappings can be applied in the master or in a slave.

Creating a Slave

To designate a Permissioning DataSource as a slave, your code must call a method that sets the role of the Permissioning DataSource as a named slave. In the following code sample, the role is set in step 3. The rest of the code is similar to the code sample described in Creating a Permissioning DataSource Adapter 7, except that a slave can only send a limited set of permissioning data to Liberator (see Master/Slave Limitations 14).

```
// Step 1 (part 1/3): Implement the PermissioningConnectionListener interface.
public class SlavePermissioningSystemAdapter implements PermissioningConnectionListener
{
   private PermissioningDataSource pdss;

   public SlavePermissioningSystemAdapter() throws IOException, SAXException
   {
      // Step 2: Instantiate a PermissioningDataSource, passing in this
      // adapter as a listener.
      pdss = new PermissioningDataSource(this,
                                         <DataSource.Config.Stream>,
                                         <Fields.Config.Stream>);

      // Step 3: Set the role of the PermissioningDataSource to slave,
      // and set the name of the slave to "FX".
      pdss.setSlaveRole("FX");

      // Step 4: In this example we apply permission data before starting
      // the PermissioningDataSource, but this can be done at any time.
      // Apply the permissioning data as part of an image transaction.
      pdss.startImageTransaction();

      // Create some Users and apply permissions and subject mappings.
      // Note: Users created here must also be created in the master.
      User user1 = pdss.createUser( ... );
      user1.applyPermission( ... );
      user1.setSubjectMapping( ... );

      User user2 = pdss.createUser( ... );
      user2.applyPermission( ... );

      // Commit the permissioning data. The data is sent to Liberator as
      // soon as a connection to Liberator is established.
      pdss.commitTransaction();

      // Step 5: Start the PermissioningDataSource.
      pdss.start();
   }

   // Step 1 (part 2/3): Implement the onConnect() callback
   ...

   // Step 1 (part 3/3): Implement the onDisconnect() callback.
   ...
}
```

In the code sample above, a slave Permissioning DataSource is created with the name "FX". In this case the slave applies permissions for two users (`user1` and `user2`). A similar piece of code could be created for the slave named "FI".

When you configure Liberator, you must also include the name of the slave in the `include-pattern` configuration option of `add-data-service` (see "Configuring Liberator to Connect to Multiple Permissioning DataSources" in **Caplin Xaqua: Installing Permissioning Components**).

Master/Slave Limitations

When permissioning data is sent to Liberator from master/slave Permissioning DataSources, the slaves can only send a limited set of permissioning data. The following table indicates the permissioning data that can be set in the master and slave Permissioning DataSources, where a "Y" indicates that data can be set and an "N" indicates that data cannot be set.

Master/Slave permissioning data limitations

Master/ Slave	Rules	Groups	User Permissions	User Password	User Attributes	Subject Mapping
Master	Y	Y	Y	Y	Y	Y
Slave	N	Y	Y	N	Y	Y

In addition to the limitations specified in the table above, users must be created in the master Permissioning DataSource before end-users can log in to Liberator. The permissions of users created in the master can then be set in a slave, as shown in the following code samples.

First create the users "John Smith" and "Fred Dibble" in the master:

```
...

// start a PermissioningDataSource update transaction
pdsm.startUpdateTransaction();

// create two Users without permissions
// Note: Users created here can be given permissions in the slave or the master
User user1 = pdsm.createUser("John.Smith", "johnsPassword");
User user2 = pdsm.createUser("Fred.Dibble", "fredsPassword");

// send the permissioning data by committing the transaction
pdsm.commitTransaction();

...
```

Now give "John Smith" and "Fred Dibble" permissions in the slave:

```
...

// start a PermissioningDataSource update transaction
pdss.startUpdateTransaction();

// create Users and apply permissions
// Note: Users created here (without passwords) must also be created
// in the master (with passwords)
User user1 = pdss.createUser("John.Smith", "");
user1.applyPermission( ... );

User user2 = pdss.createUser("Fred.Dibble", "");
user2.applyPermission( ... );

// send the permissioning data by committing the transaction
pdss.commitTransaction();

...
```

Note that the password for each user must be set in the master and not in the slave.

User Attributes and Subject Mappings

User attributes and subject mappings can be set in either the master or slave Permissioning DataSource, but you must make sure that only one Permissioning DataSource sets a particular user attribute or subject mapping.

User Attributes

If the same user attribute is set to different values in more than one Permissioning DataSource, then the value retrieved by the **Caplin Xaqua client** cannot be determined and could be either value.

For example, if the master sets `MaxUSD` to `5000` and the slave sets `MaxUSD` to `8000`, then either `5000` or `8000` could be returned when the Caplin Xaqua client retrieves the user attribute `MaxUSD`.

Subject Mappings

If a subject is mapped in more than one Permissioning DataSource, even if wildcards are used to define the subject, then it is not possible to determine what mapping will be applied by the Permissioning Auth Module.

For example, if the master maps `/FX/EURGBP` to `tier1` and the slave maps `/FX/EUR*` to `tier2`, then the Permissioning Auth Module could map a request for `/FX/EURGBP` to either `tier1` or `tier2`.

Setting the Master/Slave Roles

The following examples show you how to set the roles of the master and slave Permissioning DataSources.

Setting the Master Role

This example sets a `PermissioningDataSource` (`pdsm`) as the master Permissioning DataSource.

```
// set the master role and name the slave ("FX")
pdsm.setMasterRole();
...

pdsm.startUpdateTransaction();
...
pdsm.commitTransaction();
```

There can only be one master but there can be more than one slave Permissioning DataSource (see Setting the Slave Role 16).

The role of the master must be set before the transaction is started.

Setting the Slave Role

This example sets a `PermissioningDataSource` (`pdss`) as a slave Permissioning DataSource.

```
// set the slave role and give the slave a name ("FX")
pdss.setSlaveRole("FX");
...

pdss.startUpdateTransaction();
...
pdss.commitTransaction();
```

In this example the `setSlaveRole()` method sets the role of the Permissioning DataSource to 'slave' and names the slave "FX".

The role of the slave must be set before the transaction is started.

3.3 About Transactions

Transactions ensure that one or more operations on permissioning data are sent to Liberator as a single atomic unit. A typical sequence of events would be:

1. Start a transaction.
2. Apply permissioning data to the `PermissioningDataSource` (for example add and remove users, groups and permissions).
3. Commit the transaction.

Permissioning data is sent from the `PermissioningDataSource` to Liberator when the transaction is committed. The Permissioning Auth Module (which is embedded in Liberator) will not apply any permissioning data until all the data for a transaction is received.

API methods for starting a transaction

The Permissioning DataSource API provides two methods for starting a transaction.

- `startImageTransaction()`

  Call this method when you want to apply a new set of permissioning data to Liberator. When you commit the transaction, all permissioning data in the `PermissioningDataSource` is sent to Liberator. Liberator replaces any permissioning data from previous transactions with this new permissioning data. Rules *must* be applied as part of an image transaction.

- `startUpdateTransaction()`

  Call this method when you want to update permissioning data. When you commit the transaction, only changes to permissioning data are sent to Liberator. Liberator updates any permissioning data from previous transactions with this new permissioning data. Rules *cannot* be applied as part of an update transaction.

When should an Image or Update transaction be used?

The table below shows the type of transaction that is required (image or update) to send permissioning data to Liberator. The `startImageTransaction()` method starts an image transaction, and the `startUpdateTransaction()` method starts an update transaction (see About Transactions 16).

Situation	Type of transaction required
When you start your `PermissioningDataSource`.	Start an image transaction. The permissioning data that you send will replace any existing permissioning data in Liberator. The `PermissioningDataSource` does not send permissioning data to Liberator until the first transaction is committed. The first transaction should either be committed before the `PermissioningDataSource` is started, or as soon as it is started, as Liberator will use permissioning data from an earlier connection (if it exists).
When permissioning data in your Permissioning System changes (for example, when a new user is added to your Permissioning System).	Start an update transaction. The permissioning data that you send will modify the existing permissioning data in Liberator.
When you want to replace an existing set of permissioning data with a new set of permissioning data.	Start an image transaction. The permissioning data that you send will replace any existing permissioning data in Liberator.
When you want remove all permissioning data and eject all users currently logged in to a Caplin Xaqua client and/or Liberator.	Send an empty image transaction. This will clear all permissioning data from the `PermissioningDataSource` and from Liberator.

3.4 Creating Rules

Rules state the permissions that users must have for an **action** (see Master/Slave Limitations 14).

In this example the user must have "SPOT" **permission** for the **product** in the "Instrument" field of the RTTP message, when the subject of the RTTP message matches the regular expression "/TradeChannel/.*" and the value of the "SIDE" field is "Buy".

```
pds.startImageTransaction();

Map<String,String> fieldMatchCriteria = new HashMap<String,String>();
fieldMatchCriteria.put("SIDE","Buy");
pds.createActionRule("/TradeChannel/.*", fieldMatchCriteria, "TradeType",
                     "SPOT", "Instrument");

// add Users, Groups and Permissions for this image transaction
...

pds.commitTransaction();
```

Rules must be applied as part of an image transaction (see About Transactions 16).

3.5 Updating Permissioning Data

The following examples show you how to update the permissioning data that has already been sent to Liberator (see Master/Slave Limitations 14). You update permissioning data as part of an update transaction (see About Transactions 16).

Creating Users

This example creates a new user in the `PermissioningDataSource` (`pds`). When the transaction is committed, the data for this user is sent to Liberator.

```
pds.startUpdateTransaction();
pds.createUser("John.Smith", "johnsPassword");
pds.commitTransaction();
```

The `getUser()` method can later be used to get a reference to the user "John Smith" (see Setting a User's Password 20).

Applying Permissions

Permissions can either be applied as part of the same transaction in which the user is created, or in later transactions.

The following example creates a new user and then gives this user the permission to "SPOT-TRADE" all products in the "TradeType" namespace.

```
pds.startUpdateTransaction();
User newUser = pds.createUser("John.Smith", "johnsPassword");
Set products = new HashSet();
products.add("/.*");
newUser.applyPermission(products, "TradeType", "SPOT-TRADE", Authorization.ALLOW);
pds.commitTransaction();
```

We look at how to change the permissions of an existing user in Changing a User's Permissions 20.

Creating Groups

The following example creates a new group, applies a permission to the group, and then adds an existing user to the group. When the transaction is committed, the data for this group is sent to Liberator.

```
pds.startUpdateTransaction();

// create a new Group
Group newGroup = pds.createGroup("RFQ-Traders");

// build up a product set
Set products = new HashSet();
products.add("/.*");

// apply the permission to the Group
newGroup.applyPermission(products, "TradeType", "RFQ", Authorization.ALLOW);

// retrieve an existing user from the permissioning datasource
User existingUser = pds.getUser("John.Smith");

//add the user as a member of the new Group
newGroup.addMember(existingUser);
pds.commitTransaction();
```

In the example above, `pds.getUser()` retrieves an existing user from the `PermissioningDataSource`. This user, who was created in an earlier transaction (see Creating Users) 18, now inherits the permissions of the new group to "RFQ" trade all products in the "TradeType" namespace.

Removing Users and Groups

In this example we remove the user and group that we created in previous transactions (see Creating Users 18 and Creating Groups) 18.

```
pds.startUpdateTransaction();
Group group = pds.getGroup("RFQ-Traders");
pds.removeGroup(group);

User user = pds.getUser("John.Smith");
pds.removeUser(user);
pds.commitTransaction();
```

When you remove a group that has members, the members are not removed from the inheritance hierarchy but they no longer inherit permissions from the removed group or any of its parents.

When you remove a user, the user is automatically removed from all parent groups and will no longer be able to log in to a Caplin Xaqua client. If the removed user was already logged in to a Caplin Xaqua client, then they will be disconnected.

When you remove a user or group, references to the removed user or group object can no longer be used and should be de-referenced so that the object can be garbage collected. If you need to re-create a removed user or group, use `createUser()` or `createGroup()` inside a transaction to create a new object for that user or group.

Setting a User's Password

In this example we change a user's password.

```
pds.startUpdateTransaction();
User user = pds.getUser("John.Smith");

// set the new password
user.setPassword("new-password");
pds.commitTransaction();
```

If a user's password is changed when the user is logged in to Liberator, they will be disconnected immediately and will have to log back in using the new password.

Changing a User's Permissions

The permissions assigned to a user can be changed using the following methods:

- User.applyPermission() 20
- User.removePermission() 21
- User.permit() 21
- User.deny() 22

User.applyPermission()

This method sets a user permission that either allows or denies a single action on a product set and namespace.

In the following example, the user permission to "OneClick" trade the "FX/GBPUSD" product in the "TradeType" namespace is allowed.

```
pds.startUpdateTransaction();

// acquire a reference to the User
User user = pds.getUser("John.Smith");

// build up the product set
Set products = new HashSet();
products.add("/FX/GBPUSD");

// apply the permission
user.applyPermission(products, "TradeType", "OneClick", Authorization.ALLOW);
pds.commitTransaction();
```

This permission is added to the permissions already assigned to this user, and replaces any other permission the user has for this action, product set, and namespace.

User.removePermission()

This method removes the permission a user has for a single action on a product set and namespace.

In the following example, the user permission to "OneClick" trade the "FX/GBPUSD" product in the "TradeType" namespace is removed. We assigned this permission in the previous transaction (see User.applyPermission() 20).

```
pds.startUpdateTransaction();
User user = pds.getUser("John.Smith");
Set products = new HashSet();
products.add("/FX/GBPUSD");

// remove the OneClick permission in the TradeType namespace for /FX/GBPUSD
user.removePermission(products, "TradeType", "OneClick");
pds.commitTransaction();
```

Attempting to remove a permission that has not been assigned has no effect.

User.permit()

This method sets a user permission that allows one or more actions on a product set and namespace. The method differs from `User.applyPermission()` in that:

- Multiple actions on a product set can be allowed by a single call to this method.
- The method can only be used to allow actions, not to deny actions. To deny actions, call User.deny() 22.

Because the actions are passed in as Java `varargs`, any number of actions can be passed to `User.permit()`.

Tip: For further information about Java `varargs`, see the description from Oracle at http://download.oracle.com/javase/1,5.0/docs/guide/language/varargs.html.

In the following example, the user permission to trade the "/FX/GBPUSD" product on tenors of one week ("1W"), two weeks ("2W"), and three weeks ("3W") is allowed. The example assumes that tenor permissions are in the "Tenor" namespace, and that the name of each tenor identifies the action that must be allowed.

```
pds.startUpdateTransaction();

// acquire a reference to the User
User user = pds.getUser("John.Smith");

// build up the product set
Set products = new HashSet();
products.add("/FX/GBPUSD");

// apply the permission
user.permit(products, "Tenor", "1W", "2W", "3W");
pds.commitTransaction();
```

This permission is added to the permissions already assigned to this user, and replaces any other permission the user has for these actions, product set, and namespace.

User.deny()

This method sets a user permission that denies one or more actions on a product set and namespace. The method differs from `User.applyPermission()` in that:

- Multiple actions on a product set can be denied by a single call to this method.
- The method can only be used to deny actions, not to allow actions. To allow actions, call User.permit() 21.

Because the actions are passed in as Java `varargs`, any number of actions can be passed to `User.deny()`.

Tip: For further information about Java `varargs`, see the description from Oracle at http://download.oracle.com/javase/1,5.0/docs/guide/language/varargs.html.

In the following example, the user permission to trade the "/FX/EURUSD" product on tenors of one week ("1W"), two weeks ("2W"), and three weeks ("3W") is denied. The example assumes that tenor permissions are in the "Tenor" namespace, and that the name of each tenor identifies the action that must be denied.

```
pds.startUpdateTransaction();

// acquire a reference to the User
User user = pds.getUser("John.Smith");

// build up the product set
Set products = new HashSet();
products.add("/FX/EURUSD");

// apply the permission
user.permit(products, "Tenor", "1W", "2W", "3W");
pds.commitTransaction();
```

This permission is added to the permissions already assigned to this user, and replaces any other permission the user has for these actions, product set, and namespace.

Changing a Group's Permissions

The permissions assigned to a group can be changed using the following methods:

- Group.applyPermission() 23
- Group.removePermission() 23
- Group.permit() 23
- Group.deny() 24

Group.applyPermission()

This method sets a group permission that either allows or denies a single action on a product set and namespace.

In the following example, the group permission to "OneClick" trade all FX products in the "TradeType" namespace is allowed.

```
pds.startUpdateTransaction();

// acquire a reference to the group
Group group = pds.getGroup("JuniorTraders");

// allow "OneClick" action on all FX products
Set products = new HashSet();
products.add("/FX/.*");
group.applyPermission(products, "TradeType", "OneClick", Authorization.ALLOW);

pds.commitTransaction();
```

This permission is added to the permissions already assigned to this group, and replaces any other permission the group has for this action, product set, and namespace.

Group.removePermission()

This method removes the permission a group has for a single action on a product set and namespace.

In the following example, the group permission to "OneClick" trade all FX products in the "TradeType" namespace is removed. We assigned this permission in the previous transaction
(see Group.applyPermission() 23).

```
pds.startUpdateTransaction();

// acquire a reference to the group
Group group = pds.getGroup("JuniorTraders");

// remove "OneClick" action on all FX products
Set products = new HashSet();
products.add("/FX/.*");
group.removePermission(products, "TradeType", "OneClick");

pds.commitTransaction();
```

Attempting to remove a permission that has not been assigned has no effect.

Group.permit()

This method sets a group permission that allows one or more actions on a product set and namespace. The method differs from `Group.applyPermission()` in that:

- Multiple actions on a product set can be allowed by a single call to this method.
- The method can only be used to allow actions, not to deny actions. To deny actions, call Group.deny() 24.

Because the actions are passed in as Java `varargs`, any number of actions can be passed to `Group.permit()`.

Tip: For further information about Java `varargs`, see the description from Oracle at http://download.oracle.com/javase/1,5.0/docs/guide/language/varargs.html.

In the following example, the group permission to trade all FX products on the accounts “Jones”, “Mortimer Ltd”, and “XYZ Corp” is allowed. The example assumes that account permissions are in the "Account" namespace, and that the name of each account identifies the action that must be allowed.

```
pds.startUpdateTransaction();

// acquire a reference to the Group
Group group = pds.getGroup("AccountTraders");

// build up the product set
Set products = new HashSet();
products.add("/FX/.*");

// apply the permissions
group.permit(products, "Account", "Jones", "Mortimer Ltd", "XYZ Corp");
pds.commitTransaction();
```

This permission is added to the permissions already assigned to this group, and replaces any other permission the group has for these actions, product set, and namespace.

Group.deny()

This method sets a group permission that denies one or more actions on a product set and namespace. The method differs from `Group.applyPermission()` in that:

- Multiple actions on a product set can be denied by a single call to this method.
- The method can only be used to deny actions, not to allow actions. To allow actions, call Group.permit() 23.

Because the actions are passed in as Java `varargs`, any number of actions can be passed to `Group.deny()`.

Tip: For further information about Java `varargs`, see the description from Oracle at http://download.oracle.com/javase/1,5.0/docs/guide/language/varargs.html.

In the following example, the group permission to trade all FX products on the accounts “Walker & Baines” and “Zedco Corp” is denied. The example assumes that account permissions are in the "Account" namespace, and that the name of each account identifies the action that must be denied.

```
pds.startUpdateTransaction();

// acquire a reference to the User
Group group = pds.getGroup("John.Smith");

// build up the product set
Set products = new HashSet();
products.add("/FX/.*");

// apply the permission
group.deny(products, "Account", "Walker & Baines", "Zedco Corp");
pds.commitTransaction();
```

This permission is added to the permissions already assigned to this group, and replaces any other permission the group has for these actions, product set, and namespace.

Changing Subject Mappings for a User

A default **subject mapper** is provided with the Permissioning software that allows one subject mapping to be added for a user. If you want to add multiple subject mappings for a user, or if you want to provide customized mapping logic, then you must specify the subject mapper class that provides these mappings.

Tip: The `RegexSuffixSubjectMapper` class of the Permissioning DataSource API allows you to add multiple subject mappings for a user (see the **Permissioning DataSource: API Reference** for further information). If you want to provide customized mapping logic, then you must write a custom subject mapper class (see Creating a Custom Subject Mapper 30).

You will find further information about subject mapping in the document **Caplin Xaqua: Permissioning Overview And Concepts**.

Using the default subject mapper

With the default subject mapper, the `setSubjectMapping()` method adds a new subject mapping or changes an existing subject mapping.

The following example shows a subject mapping being changed for one user, and a subject mapping being removed for another user.

```
pds.startUpdateTransaction();

// modify User with existing subject-mapping
User userWithChangedMapping = pds.getUser("John.Smith");
userWithChangedMapping.setSubjectMapping("/FX/.*", "-tier2");

// remove a User's subject-mapping
User userWithRemovedMapping = pds.getUser("Jane.Davis");
userWithRemovedMapping.removeSubjectMapping();
pds.commitTransaction();
```

Because a user can only have one subject mapping, the `removeSubjectMapping()` method does not require any parameters.

Attempting to remove a subject mapping that has not been assigned has no effect.

Specifying the subject mapper

If you want to provide multiple or customized subject mappings for a user, the `setSubjectMapper()` method specifies the class of the subject mapper that you want to use, and the `addSubjectMapping()` method adds subject mappings for that user.

The following example maps prices for FX and FI instruments. The example assumes that a custom subject mapper has been created and that Liberator has been configured to use this custom subject mapper.

```
pds.startUpdateTransaction();

// specify the User
User userWithCustomMapping = pds.getUser("Pauline.Jones");

// specify the class that implements the custom subject mapper for this User
userWithCustomMapping.setSubjectMapper("com.mydomain.MyCustomSubjectMapper");

// add some subject mappings for FX trades
Map<String,String> fxMappings = new HashMap<String,String>();
fxMappings.put("USDGBP","-tier1");
fxMappings.put("USDEUR","-tier2");
userWithCustomMapping.addSubjectMapping("FX", fxMappings);


// add some subject mappings or FI trades
Map<String,String> fiMappings = new HashMap<String,String>();
fiMappings.put("DEFAULT","-tier1");
fiMappings.put("ORCL","-tier2");
fiMappings.put("MSFT","-tier3");
userWithCustomMapping.addSubjectMapping("FI", fiMappings);

pds.commitTransaction();
```

In this example the prices shown to the user will be from tier 1, tier 2, or tier 3, depending on the instrument requested. Note that `addSubjectMapping()` adds a subject mapping when you specify the subject mapper, but `setSubjectMapping()` adds a subject mapping when you are using the default subject mapper.

To remove subject mappings from a custom subject mapper, call `setSubjectMapper()` as part of an update transaction. When this method is called a new instance of the subject mapper is created with no mappings (effectively removing existing mappings).

Updating the global context of Subject Mappers

The **global context** is an object at the Permissioning Auth Module and contains data that any subject mapper can access. In this way a custom subject mapper can map subjects using logic based on this common data, and not just on subject mappings defined for the user.

For example, if a custom subject mapper uses FX rates to map a subject, it is a more efficient use of memory and bandwidth to add these rates to the global context than to the subject mappings of every user.

Data for the global context is sent to the Permissioning Auth Module by the Permissioning DataSource.

Note: There is only one global context and it is shared by all Permissioning DataSources. Therefore if you have multiple Permissioning DataSources, make sure they do not overwrite each other's data.

A default global context class is provided with the Permissioning software, but you can also create your own class that provides additional methods to subject mappers. To create and deploy a custom global context, see Creating a Custom Global Context 35.

Adding data to the global context

To add data to the global context, call the `updateGlobalContext()` method as part of a transaction. The following example adds FX rates to the global context, and sets the identifier of this data to "RATES".

```
pds.startUpdateTransaction();

  final Map<String,String> ratesData = new HashMap<String,String>();
  ratesData.put("USDGBP", "1.34");
  ratesData.put("USDEUR", "1.41");

  // add data to the global context
  pds.updateGlobalContext("RATES", ratesData);

pds.commitTransaction();
```

A custom subject mapper can now use these FX rates in the logic that maps a subject for a user.

Tip: Because the arguments to `updateGlobalContext(String, Map)` are the same as the arguments to `addSubjectMapping(String, Map)`, it is relatively simple to migrate data, such as FX rates, from a user's subject mapper to the global context.

An example custom subject mapper that accesses global context data is shown in Creating a Custom Global Context 35.

Removing data from the default global context

To remove data from the default global context, pass the identifier of the data that you want to remove to the `removeGlobalContextData()` method. The following example removes "RATES" from the default global context.

```
pds.startUpdateTransaction();

  // remove data from the global context
  pds.removeGlobalContextData("RATES");

pds.commitTransaction();
```

Changing User Attributes

A user can be assigned any number of attributes in the form of name/value pairs.

In this example we change the value of the "MaxTradeDollars" attribute to 3 million for an existing user.

```
pds.startUpdateTransaction();
User user = pds.getUser("John.Smith");

// modify an existing attribute (assumes MaxTradeDollars already set - not shown here)
user.setAttribute("MaxTradeDollars", "3000000");
pds.commitTransaction();
```

The next example shows how to remove the "MaxTradeDollars" attribute from the same user.

```
pds.startUpdateTransaction();
User user = pds.getUser("John.Smith");

// remove an attribute
user.removeAttribute("MaxTradeDollars");
pds.commitTransaction();
```

Attempting to remove an attribute that has not been assigned has no effect.

Changing the Members of a Group

The members of a group can be changed using the methods `Group.addMember()` and `Group.removeMember()`. Adding and removing group members affects every child that inherits from the group.

In this example we give an existing user a new parent and grandparent.

```
pds.startUpdateTransaction();
User user = pds.getUser("John.Smith");

// create the parent Group and add the User as a member
Group parent = pds.createGroup("Parent");
parent.addMember(user);

// create the grandparent group and add the earlier parent group as a member
Group grandparent = pds.createGroup("Grandparent");
grandparent.addMember(parent);

pds.commitTransaction();
```

The user will now inherit permissions (not shown in this example) from both the parent and the grandparent.

We now remove the parent group from the grandparent group.

```
pds.startUpdateTransaction();

// acquire a reference to the two groups that are to be detached from each other
Group parent = pds.getGroup("Parent");
Group grandparent = pds.getGroup("Grandparent");

// sever the relationship
grandparent.removeMember(parent);
pds.commitTransaction();
```

The user continues to inherit permissions from the parent group but no longer inherits permissions from the grandparent group, because the grandparent is no longer an ancestor of this user.

4 Creating a Custom Subject Mapper

Subject mapping allows the subject of an RTTP message to be modified by Liberator. Subject mapping is transparent to the user and could be used, for example, to provide preferential data to selected users (see **Caplin Xaqua: Permissioning Overview And Concepts** for further details).

Subjects are modified in the Permissioning Auth Module from mappings that you set in the Permissioning DataSource. For example the subject "FX/USDGBR" could be changed to "FX/USDGBR-tier2", so that the end-user is shown tier 2 prices when they request the "FX/USDGBR" instrument.

The default subject mapper provided with the Permissioning software allows one subject mapping to be added for a user, and the `RegexSuffixSubjectMapper` class of the Permissioning DataSource API allows multiple subject mappings to be added. If you want to calculate subject mappings based on your own mapping logic, then you must create a custom subject mapper.

To create a custom subject mapper you must:

- Write custom Java code that implements the SubjectMapper Interface 30 of the Permissioning DataSource API.
- Compile the custom Java code and deploy it to the Permissioning Auth Module 33.

4.1 Implementing the SubjectMapper Interface

When you create a custom subject mapper, the Java code that you write must implement the `SubjectMapper` interface of the Permissioning DataSource API. This interface has three methods.

- `updateMappings(String key, Map<String, String> updateMap)`

  This method is called by the Permissioning Auth Module when subject mappings are received from the Permissioning DataSource. The method is passed a key and the subject mappings for that key.

  The key and subject mappings are set in the Permissioning DataSource using the `User.setSubjectMapper()` and `User.addSubjectMappings()` methods, and sent to the Permissioning Auth Module as part of a transaction.

  The `updateMappings()` method has no return value but allows you to store the received keys and subject mappings, and to make them available to `mapSubject()`. Each subject mapping typically consists of a subject pattern and subject suffix, and the key associated with the mapping.

- `mapSubject(String subject)`

  This method is called by the Permissioning Auth Module when Liberator receives an RTTP message from the client application. The `subject` passed to this method is the subject of the RTTP message received by Liberator.

  If a mapping exists for this subject, the method must return the modified subject as a string. Liberator uses the modified subject to communicate with the DataSource and to check user permissions.

  If a mapping does not exist for this subject, the method must return return null. In this case Liberator uses the original subject to communicate with the DataSource and to check user permissions.

- `setGlobalContext(GlobalContext globalContext)`

  This method is called by the Permissioning Auth Module when the global context is updated by the Permissioning DataSourrce as part of a transaction (see Setting the global context for Subject Mappers 26). The `globalContext` object that is passed to this method contains all global context data, and not just the updated data.

If the subject mapper needs to access global context data when the Permissioning Auth Module calls `mapsubject()`, for example in the logic that maps a subject, the method must store a reference to `globalContext`. If the subject mapper does not need to access global context data, the method can be empty and simply return. This method has no return value.

The `SubjectMapper` interface that you implement must either provide a default (no argument) constructor, or let the compiler create the default constructor. A default constructor is required so that instances of the custom `SubjectMapper` class can be created dynamically.

Example Implementation of SubjectMapper

The following is an example of a custom subject mapper that implements the `SubjectMapper` interface of the Permissioning DataSource API. Comments in the example describe how it works.

Tip: This example provides the same methods as the `RegexSuffixSubjectMapper` class of the Permissioning DataSource API, which is provided with the Permissioning software kit.

```
package example.mapper;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.regex.Pattern;
import com.caplin.permissioning.SubjectMapper;
import com.caplin.permissioning.GlobalContext;

public class CustomRegexSubjectMapper implements SubjectMapper
{
  // Stores the mapping data in a list of PatternSuffixPairs. PatternSuffixPairs
  // associate regex patterns with the suffixes that should be appended if a given
  // regex pattern matches.
  private final List<PatternSuffixPair> patternsToSuffixes =
          new ArrayList<PatternSuffixPair>();

  // This method is called by the Permissioning Auth Module when it receives an RTTP
  // message from the client application. This method must map the subject if a mapping
     exists for this subject, otherwise it should return null.
  //
  // In this implementation we attempt to match the passed in subject with each regex
  // in turn. If a regex matches the passed in subject, we return a new string that
  // is constructed by concatenating the suffix associated with the matching regex to
  // the passed in subject. If no regexs match, we return null to signify that the
  // subject has not been mapped.
  private String mapSubject(String subject)
  {
    for(PatternSuffixPair pair : patternsToSuffixes)
    {
      if(pair.pattern.matcher(subject).matches())
      {
        return subject+pair.suffix;
      }
    }
    return null;
  }

  // This method is called by the Permissioning Auth Module when global context data
  // is received from the Permissioning DataSource. As the global context is not used
  // by this SubjectMapper implementation, this method does not do anything.
  public void setGlobalContext(GlobalContext globalContext)
  {
    // no-op
  }
  // This method is called by the Permissioning Auth Module when subject mapping data
  // is received from the Permissioning DataSource.
  //
  // This method must store the passed in data so that it can be used in the
  // implementation of mapSubject(String). Note: when storing the data, we
  // preemptively compile the regex to avoid having to do so repeatedly in each
  // call to mapSubject(String).
  public void updateMappings(String identifier, Map<String,String> updateMap)
  {
    for(Map.Entry<String,String> entry : updateMap.entrySet())
    {
      final Pattern pattern = Pattern.compile(entry.getKey());
      final String suffix = entry.getValue();
      patternsToSuffixes.add(new PatternSuffixPair(pattern, suffix));
    }
  }

  // This simple class is used to associate the regex patterns compiled by
  // updateMappings() with their suffixes.
  private static class PatternSuffixPair
  {
    public final Pattern pattern;
    public final String suffix;

    public PatternSuffixPair(Pattern pattern, String suffix)
    {
      this.pattern = pattern;
      this.suffix = suffix;
    }
  }
}
```

The following is an example of how the custom subject mapper shown above could be used, after it has been deployed to the Permissioning Auth Module.

At the Permissioning DataSource

The Permissioning DataSource sends two subject mappings to the Permissioning Auth Module as part of a transaction. The first subject mapping maps the regular expression (regex) "/FX/USD.*" to the string "-tier1", and the second maps the regex "/FX/EUR.*" to the string "–tier2". The identifier for these subject mappings is "FXkey1".

```
permissioningDataSource.startUpdateTransaction();

   // define the subject mappings
   final Map<String,String> mappingData = new HashMap<String,String>();
   mappingData.put("/FX/USD.*", "-tier1");
   mappingData.put("/FX/EUR.*", "-tier2");

  // the subject mappings will be added for this user
   final User existinguser = permissioningDataSource.getUser(username);

   // set the subject mapper for this user
   user.setSubjectMapper(CustomRegexSubjectMapper.class.getName());

   // add the subject mappings for this user
   user.addSubjectMapping("FXkey1", mappingData);

 permissioningDataSource.commitTransaction();
```

At the Permissioning Auth Module

When the transaction for this subject mapping is received from the Permissioning DataSource, the Permissioning Auth Module calls the interface method `updateMappings()`, passing in the received identifier and subject mappings. The `updateMappings()` method iterates over the passed in subject mapping pairs, and saves each key-value pair as a compiled regex pattern and associated subject suffix.

When the user requests an instrument, the Permissioning Auth Module calls `mapSubject()`, passing in the subject of the RTTP message request. If this subject matches a saved regex pattern, `mapSubject()` concatenates the suffix for this pattern to the passed in subject, and returns the concatenated string. If the subject of a message does not match a saved regex pattern, `mapSubject()` returns null.

For example, if the user requests "/FX/USDGBP", `mapSubject()` matches this to the regex pattern "/FX/USD.*" and returns "/FX/USDGBP-tier1" as the mapped subject. Alternatively, if the user requests "/FX/EURAUD", `mapSubject()` matches this to the regex pattern "/FX/EUR.*" and returns "/FX/EURAUD-tier2" as the mapped subject.

Although the subject mapper in this example does not use the passed in identifier "FXkey1", other subject mappers could use this identifer in the logic that maps a subject.

4.2 Deploying a custom Subject Mapper

If you create a custom subject mapper that implements the `SubjectMapper` interface of the Permissioning DataSource API, then you must deploy the compiled class file, or a JAR file containing the compiled class, to a classpath of the Permissioning Auth Module. To deploy the compiled subject mapper class:

- Copy the class or JAR file to a directory that Liberator can access.
- Add the directory as a classpath in the Liberator configuration file *java.conf*.

Deploying Class Files to the Permissioning Auth Module

Class files are typically copied to */lib/java* in the Liberator installation directory, and in a directory structure that corresponds to the package location. When you have copied the class file, add the classpath for this directory to the Liberator configuration file *java.conf*.

```
add-javaclass
    class-name   com.caplin.permissioning.PermissioningAuthModule
    class-id     authenticator
    classpath    %r/../kits/permissioning-auth-module-latest-jar-
                           with-dependencies.jar
    classpath    %r/lib/java/
end-javaclass
```

In the example configuration above, **`%r`** is a symbolic reference to the Liberator installation directory.

Deploying JAR Files to the Permissioning Auth Module

JAR files are typically copied directly to */lib/java* in the Liberator installation directory. When you have copied the JAR file, add the classpath for the JAR file to the Liberator configuration file *java.conf*.

```
add-javaclass
    class-name   com.caplin.permissioning.PermissioningAuthModule
    class-id     authenticator
    classpath    %r/../kits/permissioning-auth-module-latest-jar-
                            with-dependencies.jar
    classpath    %r/lib/java/MyCustomSubjectMapper.jar
end-javaclass
```

In the example configuration above, **`%r`** is a symbolic reference to the Liberator installation directory.

5 Creating a Custom Global Context

The global context is an object at the Permissioning Auth Module that implements the `GlobalContext` interface. This interface allows a custom subject mapper to access data that is common to all subject mappers and users. In this way a subject mapper can map subjects using logic based on this common data, and not just on subject mappings defined for the user.

For example, if a custom subject mapper uses FX rates to map a subject, it is a more efficient use of memory and bandwidth to add these rates to the global context than to the subject mappings of every user.

A default implementation of `GlobalContext` is provided with the Permissioning software, and provides a `get()` method for accessing this common data. You can also write a custom class that implements this interface, or extend the default implementation to provide additional methods that subject mappers can call.

For example, a custom implementation of `GlobalContex` could provide complex objects that would otherwise require the subject mapper to make multiple `get()` calls to the default implementation. Another example is a custom implementation that provides the same complex object to several subject mappers, reducing the processing required by each subject mapper.

Data for the global context is sent to the Permissioning Auth Module by the Permissioning DataSource.

Note: There is only one global context object at the Permissioning Auth Module, and it is shared by all Permissioning DataSources. Therefore if you have multiple Permissioning DataSources, make sure they do not overwrite each other's data.

To create a custom global context you must:

1. Write a custom global context class that implements the GlobalContext Interface 36 of the Permissioning DataSource API. This class can either implement the `GlobalContext` interface directly, or extend the `DefaultGlobalContext` class that already implements this interface.
2. Write a custom subject mapper that calls the `GlobalContext` interface to access global context data (see the example subject mapper in Example Implemenation of GlobalContext 36).
3. Compile the custom code and deploy it at the Permissioning Auth Module
(see Deploying a custom Global Context 40 and Deploying a custom Subject Mapper 33).
4. Configure the Permissioning Auth Module to use your custom global context class
(see Configuring the Permissioning Auth Module 41).

5.1 Implementing the GlobalContext Interface

When you create a custom global context class, the Java code that you write must implement the `GlobalContext` interface of the Permissioning DataSource API. This interface has four methods.

- `void update(String identifier, Map<String,String> data)`

  This method is called by the Permissioning Auth Module when data for the global context is received from the Permissioning DataSource. The method is passed a Map of the data and an identifier for that data.

  The identifier and data Map are set in the Permissioning DataSource using the `PermissioningDataSource.updateGlobalContext()` method, and sent to the Permissioning Auth Module as part of a transaction.

  The method has no return value, but allows you to save the identifier and data, and to make them available to subject mappers that call the `get()` method of this interface.

- `Map get(String identifier)`

  This method is called by subject mappers that want to get the Map of data that is saved for the passed in identifier. The subject mapper can then use data from this Map in the logic that maps a subject.

- `String get(String identifier, String key)`

  This method is called by subject mappers that want to get the data value that is saved for the passed in identifier and key. The subject mapper can then use the data value in the logic that maps a subject. Calling this method is the same as calling `get(identifier)`, and then calling `get (key)` on the returned data Map.

- `void remove(String identifier)`

  This method is called by the Permissioning Auth Module when the Permissioning DataSource calls `removeGlobalContextData()` as part of a transaction. The method has no return value but must remove from the global context, the data Map for the passed in identifier.

The `GlobalContext` interface that you implement must either provide a default (no argument) constructor, or let the compiler create the default constructor. A default constructor is required so that instances of the custom `GlobalContext` class can be created dynamically.

Example Implementation of GlobalContext

The following is an example of a custom global context class that allows subject mappers to map subjects in a way that provides a 24-hour market for instruments. This implementation of the `GlobalContext` interface extends the `DefaultGlobalContext` class, and comments in the code describe in detail how it works.

The `getCurrentMarket()` method of this class gets the current time and the opening times of three markets, and uses this information to return a string that indicates the currently open market. A subject mapper that calls this method can then insert the returned string in subjects that it maps, so that instrument requests can be routed to the market that is currently open (such as the current market for indicative FX prices).

Market opening times are sent from the Permissioning DataSource and saved to the global context. The `getCurrentmarket()` method accesses this global context data when it constructs the string for the currently open market. In this way individual subject mappers do not need to access or process global context data directly.

An example of a custom subject mapper that uses this class to provide a 24-hour market is also described.

```
package example.mapper;
import java.util.Calendar;
import java.util.Map;
import java.util.SortedMap;
import java.util.TimeZone;
import java.util.TreeMap;
import com.caplin.permissioning.DefaultGlobalContext;

public class MarketRoutingGlobalContext extends DefaultGlobalContext
{
  // The identifier used by the Permissioning DataSource to indicate that updates
  // to the global context contain market opening times.
  private static final String MARKETS = "MARKETS";

  // The timezone that market opening times are expressed in.
  private static final TimeZone GMT = TimeZone.getTimeZone("GMT");

  // Store the market opening times in a format that can be easily
  // used at runtime. A SortedMap is used as its headMap(String) method
  // returns the most recent market start time (as used in getCurrentMarket())
  private final SortedMap<String,String> hoursToMarkets =
                                       new TreeMap<String,String>();
  @Override
  public void update(String identifier, Map<String, String> data)
  {
    // This class is only interested in MARKETS data.
    if(MARKETS.equals(identifier))
    {
      // Process and save the opening time of each market.
      // Opening times are sent from the Permissioning DataSource as integers,
      // for example 1 represents an opening time of 01:00, 9 an opening time
      // of 09:00, and 17 an opening time 17:00.
      for(Map.Entry<String,String> entry : data.entrySet())
      {
        // Get the market opening time as an hour between 0 and 23 (inclusive).
        final String marketStart = entry.getKey();
        // Get the string that identifies the market.
        final String market = entry.getValue();

        // Save market opening times in the format required by getCurrentMarket().
        // That is, save 1 as 0059, 9 as 0859, and so on. This is because
        // getCurrentMarket() calls SortedMap.headMap(key) to get markets that are open,
        // and headMap(key) only returns keys that are less than the passed in key
        // (otherwise the market that opens at 17:00 would not be returned until 17:01).
        final Integer previousHour = ((Integer.valueOf(marketStart) + 24) -1) %24 ;
        hoursToMarkets.put(padZeroes(previousHour)+"59", market);
      }
      // Get the market that is open at midnight, and save the key to this
      // market as "0".
      final String lastMarketStartTime = hoursToMarkets.lastKey();
      hoursToMarkets.put("0", hoursToMarkets.get(lastMarketStartTime));
    }
    // If not MARKETS data, update the superclass.
    else
    {
      super.update(identifier, data);
    }
  }
  @Override
  public void remove(String identifier)
  {
    // We only store data for the MARKETS identifier, so we
    // only have to clear data with this identifier.
    if(MARKETS.equals(identifier))
    {
      hoursToMarkets.clear();
    }
    // If not MARKETS data, the superclass must remove the data.
    else
    {
      super.remove(identifier);
    }
  }
  // A utility method that converts an hour or minute expressed as a single
  // digit integer, to an hour or minute expressed as a two-digit string.
  private String padZeroes(int hrormin)
  {
    // Call padZeroes(String)
    return padZeroes(""+hrormin);
  }

  // A utility method that inserts a leading zero in a single digit string.
  private String padZeroes(String hrormin)
  {
    while(hrormin.length() < 2)
    {
      hrormin = "0"+hrormin;
    }
    return hrormin;
  }

  // This is the custom method that SubjectMappers call to
  // find the market that is currently open.
  public String getCurrentMarket()
  {
    // Get the current hour and minute of the day.
    final Calendar now = Calendar.getInstance(GMT);
    final int currentHour = now.get(Calendar.HOUR_OF_DAY);
    final int currentMinute = now.get(Calendar.MINUTE);
    // Format the current time into a string of the form 0959.
    final String timeToLookup = padZeroes(currentHour)+padZeroes(currentMinute);

    // Get an hoursToMarkets map that contains all market opening times
    // that are earlier than the current time.
    final SortedMap<String,String> ealierMarkets =
                              hoursToMarkets.headMap(timeToLookup);
    // Return the market that opened most recently (the currently open market).
    return ealierMarkets.get(ealierMarkets.lastKey());
  }
}
```

Adding market opening times at the Permissioning DataSource

To add market opening times to the custom global context, call `updateGlobalContext()` as part of a transaction. The following example assumes that the custom `MarketRoutingGlobalContext` class has already been deployed at the Permissioning Auth Module (see Deploying a custom Global Context 40), that the Permissioning Auth Module has been configured to use this custom class (see Configuring the Permissioning Auth Module 41), and that the following markets are available:

- Hong Kong Stock exchange (HKSE), open between 01:00 GMT and 09:00 GMT.
- London Stock Exchange (LSE), open between 09:00 GMT and 17:00 GMT.
- New York Stock Exchange (NYSE), open between 17:00 GMT and 01:00 GMT the next day.

```
permissioningDataSource.startUpdateTransaction();

  final Map<String,String> marketsData = new HashMap<String,String>();
  marketsData.put(“1”, “HKSE”);
  marketsData.put(“9”, “LSE”);
  marketsData.put(“17”, “NYSE”);

  permissioningDataSource.updateGlobalContext(“MARKETS”, marketsData);

permissioningDataSource.commitTransaction();
```

An example custom subject mapper that provides a 24-hour market

The following is an example of a custom subject mapper that uses the custom `MarketRoutingGlobalContext` class to provide a market for instruments that is open 24-hours a day.

The custom subject mapper extends the example `CustomRegexSubjectMapper` class described in this document (see Example Implementation of SubjectMapper 31), and prepends mapped subjects with a string that identifies the currently open market. To get this string, the subject mapper calls the `getCurrentMarket()` method of `MarketRoutingGlobalContext`.

```
package example.mapper;
import com.caplin.permissioning.GlobalContext;

public class MarketRoutingSubjectMapper extends CustomRegexSubjectMapper
{
  // Create a reference to the custom GlobalContext
  private MarketRoutingGlobalContext marketContext;

  // Because the setGlobalContext() method of CustomRegexSubjectMapper is empty
  // and does not store the global context, override this method and
  // store the global context here.
  @Override
  public void setGlobalContext(GlobalContext globalContext)
  {
    if(globalContext instanceof MarketRoutingGlobalContext)
    {
      marketContext = (MarketRoutingGlobalContext)globalContext;
    }
  }

  @Override
  public String mapSubject(String subject)
  {
    // Use mapObject() of the super class to determine if the passed in subject
    // matches a regex subject mapping, and is therefore mapped by this subject mapper.
    String superResult = super.mapSubject(subject);
    if(superResult != null)
    {
      // If the subject is mapped by the superclass, map it further by
      // prepending the string that identifies the open market
      return "/"+marketContext.getCurrentMarket()+superResult;
    }
    else
    {
      // If the passed in subject is not mapped by this subject mapper, do not
      // prepend a market string and simply return null
      return null;
    }
  }
}
```

Adding subject mappings at the Permissioning DataSource

The following example sets the custom `MarketRoutingSubjectMapper` class as the class that maps subjects for a user, and then adds two subject mappings for that user. The example assumes that the custom `MarketRoutingSubjectMapper` class has already been deployed at the Permissioning Auth Module (see Deploying a custom Subject Mapper 33).

```
permissioningDataSource.startUpdateTransaction();

  // Define subject mappings.
  final Map<String,String> fxMappingData = new HashMap<String,String>();
  fxMappingData.put("/FX/PRICES/USDGBP", "-tier1");
  fxMappingData.put("/FX/PRICES/USDEUR", "-tier2");

  // Get the user.
  final User existinguser = permissioningDataSource.getUser(username);

  // Set the subject mapper and add subject mappings for this user.
  user.setSubjectMapper("example.mapper.MarketRoutingSubjectMapper");
  user.addSubjectMapping("any-value", fxMappingData);

permissioningDataSource.commitTransaction();
```

5.2 Deploying a custom Global Context

If you create a custom global context that implements the `GlobalContext` interface of the Permissioning DataSource API, then you must deploy the compiled class file, or a JAR file containing the compiled class, to a classpath of the Permissioning Auth Module. To deploy the compiled global context class:

- Copy the class or JAR file to a directory that Liberator can access.
- Add the directory as a classpath in the Liberator configuration file *java.conf*.

The Permissioning Auth Module must also be configured to use the custom global context (see Configuring the Permissioning Auth Module 41).

Deploying Class Files to the Permissioning Auth Module

Class files are typically copied to */lib/java* in the Liberator installation directory, and in a directory structure that corresponds to the package location. When you have copied the class file, add the classpath for this directory to the Liberator configuration file *java.conf*.

```
add-javaclass
    class-name   com.caplin.permissioning.PermissioningAuthModule
    class-id     authenticator
    classpath    %r/../kits/permissioning-auth-module-latest-jar-
                             with-dependencies.jar
    classpath    %r/lib/java/
end-javaclass
```

In the example configuration above, **`%r`** is a symbolic reference to the Liberator installation directory.

Deploying JAR Files to the Permissioning Auth Module

JAR files are typically copied directly to */lib/java* in the Liberator installation directory. When you have copied the JAR file, add the classpath for the JAR file to the Liberator configuration file *java.conf*.

```
add-javaclass
    class-name    com.caplin.permissioning.PermissioningAuthModule
    class-id      authenticator
    classpath     %r/../kits/permissioning-auth-module-latest-jar-
                             with-dependencies.jar
    classpath     %r/lib/java/MyCustomGlobalContext.jar
end-javaclass
```

In the example configuration above, **`%r`** is a symbolic reference to the Liberator installation directory.

5.3 Configuring the Permissioning Auth Module

To configure the Permissioning Auth Module to use a custom global context class, set the property `GlobalContextClass` to the fully qualified package name of the custom class in the properties file *javaauth.properties*. The following example sets this property for the `MarketRoutingGlobalContext` class.

```
GlobalContextClass=example.mapper.MarketRoutingGlobalContext
```

The *javaauth.properties* file configures the Permissioning Auth Module, and may have been supplied with the Liberator kit or created when Liberator was installed.

If *javaauth.properties* does exist, it will be located in a directory that Liberator can access. This could be:

1. In a directory defined by a `classpath` of the Permissioning Auth Module in the Liberator configuration file *java.conf*.

2. In a location referred to by a symbolic link. The symbolic link would also be located in a directory defined by a `classpath` of the Permissioning Auth Module.

If *javaauth.properties* does not exist, you will need to create it and deploy it to a classpath of the Permissioning Auth Module.

Note: Only one copy of *javaauth.properties* must be deployed, otherwise the configuration that is applied cannot be determined.

Deploying the javaauth.properties file

To deploy *javaauth.properties*:

- Copy the file to a directory that Liberator can access.
- In the Liberator configuration file *java.conf*, add the directory as a classpath of the Permissioning Auth Module.

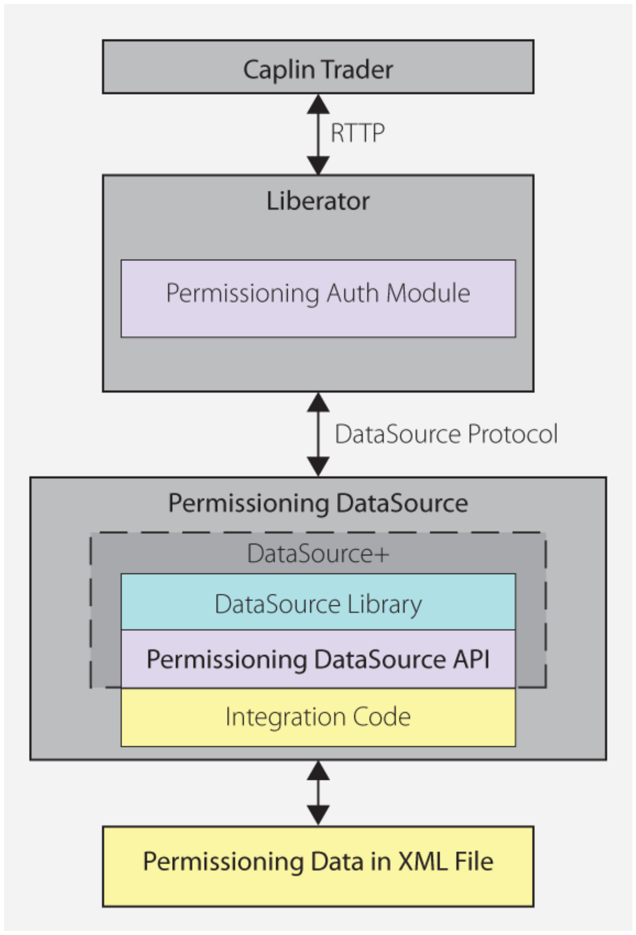
The following example shows what the Liberator configuration file *java.conf* would look like if the *javaauth.properties* file is copied to the directory *%r/etc*, and if the custom global context class `MarketRoutingGlobalContext` is deployed in the JAR file *%r/lib/java/MyCustomGlobalContext.jar* (see Deploying a custom Global Context 40).

```
add-javaclass
    class-name   com.caplin.permissioning.PermissioningAuthModule
    class-id     authenticator
    classpath    %r/../kits/permissioning-auth-module-latest-jar-
                            with-dependencies.jar
    classpath    %r/lib/java/MyCustomGlobalContext.jar
    classpath    %r/etc/
end-javaclass
```

In the example configuration above, **`%r`** is a symbolic reference to the Liberator installation directory.

6 The Demo Permissioning DataSource

The **Demo Permissioning DataSource** is an example of a Permissioning DataSource application that gets its permissioning data from an XML file. The application sends the permissioning data to Liberator when a connection to Liberator is established.



Demo Permissioning DataSource and XML File

From Caplin Trader release 1.2.8 onwards, the reference implementation of Caplin Trader is installed with a Permissioning Auth Module and Demo Permissioning DataSource example application. If you have an earlier release of Caplin Trader, then you must manually install and configure these components before you start using the Demo Permissioning DataSource (see **Caplin Xaqua: Installing Permissioning Components** for further information).

6.1 Starting and Stopping the Demo Permissioning DataSource

The Demo Permissioning DataSource is supplied with scripts that you can run to start and stop the example application.

Starting the Demo Permissioning DataSource

To start the Demo Permissioning DataSource, navigate to the *apps/caplin/PermissioningDataSource* directory and run the following command.

```
$ ./start.sh
```

This starts the application, passing the following files as arguments.

- *conf/Permissions.xml* (permissioning data in XML format)
- *conf/DataSource.xml* (DataSource configuration file)
- *conf/Fields.xml* (DataSource field mapping file)

When a connection to Liberator is established, the Demo Permissioning DataSource sends the permissioning data to Liberator.

Stopping the Demo Permissioning DataSource

To stop the Demo Permissioning DataSource, navigate to the *apps/caplin/PermissioningDataSource* directory and run the following command.

```
$ ./stop.sh
```

This stops the application and terminates the connection with Liberator.

6.2 Overview of the Demo Permissioning DataSource

The Demo Permissioning DataSource consists of one interface and two classes.

`PermissionsLoader`: This interface defines a service that loads permissioning data from a permissioning system.

`XMLPermissionsLoader`: This class implements the `PermissionsLoader` interface to load permissioning data into the `PermissioningDataSource` from the file *conf/Permissions.xml*.

`DemoPermissioningDataSource`: This class is initialized with an `XMLPermissionsLoader`. It creates a `PermissioningDataSource` to send the permissioning data to Liberator when a connection to Liberator is established. The class implements the `PermissioningConnectionListener` interface of the Permissioning DataSource API. The principal methods of the class are summarized below.

- `main(String[] args)`

  Creates the `DemoPermissioningDataSource` using the passed in arguments, retrieves permissioning data from the permissioning system, and starts the `DemoPermissioningDataSource`.

- `onConnect()`

  Called by the `PermissioningDataSource` when a Liberator connection is established. This implementation simply logs a connection established message.

- `onDisconnect()`

  Called by the `PermissioningDataSource` when a Liberator connection is lost. This implementation simply logs a connection lost message.

- `terminate()` Shuts down the `DemoPermissioningDataSource`.

You will find fully commented source code for the Demo Permissioning DataSource in *apps/caplin/kits/ permissioning-datasource-<version>/example-application* (where <*version*> = version number).

Tip: The `PermissioningConnectionListener` interface and `PermissioningDataSource` class are described in the **Permissioning DataSource: API Reference**.

7 The Demo Permissioning XML

The Demo Permissioning DataSource gets its permissioning data from an XML file, and then sends that permissioning data to Liberator when a connection to Liberator is established. This part of the document describes the XML-based elements that define the structure and content of this permissioning data.

If you want to experiment with the demo by adding or modifying permissioning data for users, groups, or rules, then you must edit the file *apps/caplin/PermissioningDataSource/conf/Permissions.xml*.

The Demo Permissioning DataSource is a master Permissioning DataSource and does not have any slaves. If you create a slave Permissioning DataSource 11 that also gets its permissioning data from XML, then you will need to create a separate XML file containing the permissioning data for that slave.

7.1 Technical Assumptions and Restrictions

XML

The XML markup defined here conforms to XML version 1.0 and the XML schema version defined at http://www.w3.org/2001/XMLSchema.

7.2 Ordering and Nesting of Tags

Each top level tag is shown below, together with the child tags that it can contain.

Tip: Advanced users may wish to consult the Relax NG Schema (*Permissions.rnc*) for definitive information on the ordering and nesting of tags. This file is supplied with the permissioning software.

For a description of each tag and its attributes, see the XML Reference Information 49 section.

<permissioning>

This is the outermost tag.

```
<permissioning>
   <rules></rules> (zero or one)
   <users></users> (zero or one)
   <groups></groups> (zero or one)
   <role></role> (zero or one)
</permissioning>
```

<rules>

```
<rules>
   <rule></rule> (one or more)
</rules>
```

<users>

```
<users>
   <user></user> (one or more)
</users>
```

<groups>

```
<groups>
   <group></group> (one or more)
</groups>
```

<role>

```
<role> (must contain only one of the following)
   <master />
   <slave />
</role>
```

<rule>

```
<rule>
   <fieldMatchCriteria></fieldMatchCriteria>  (zero or one)
</rule>
```

<user>

```
<user> (children in any order)
   <subjectMapping /> (zero or one)
   <attributes></attributes> (zero or one)
   <permissionSet></permissionSet> (zero or one)
</user>
```

<group>

```
<group>
   <permissionSet></permissionSet> (zero or one)
   <members></members> (zero or one)
</group>
```

<fieldMatchCriteria>

```
<fieldMatchCriteria>
   <match /> (one or more)
</fieldMatchCriteria>
```

<attributes>

```
<attributes>
   <userAttribute /> (one or more)
</attributes>
```

<permissionSet>

```
<permissionSet>
   <productPermissionSet></productPermissionSet> (one or more)
</permissionSet>
```

<members>

```
<members>
   <userRef /> (zero or more)
   <groupRef /> (zero or more)
</members>
```

<productPermissionSet>

```
<productPermissionSet>
   <permission /> (one or more)
</productPermissionSet>
```

<groupRef> (no children)

<match> (no children)

<master> (no children)

<slave> (no children)

<subjectMapping> (no children)

<userAttribute> (no children)

<userRef> (no children)

7.3 XML Reference Information

The following sections describe the Permissioning XML tags. They are arranged in alphabetical order of tag name.

For each tag the attributes you can use within it are listed and described in a table. The "Req?" column indicates whether the attribute is always required ("Y") or is optional ("N"). If you do not supply an optional attribute within an instance of the tag then the runtime behavior will be according to the default value of the attribute.

<attributes>

`<attributes>`

A collection of one or more user attributes, with one attribute per child <userAttribute> tag.

Attributes: This tag has no attributes.

<fieldMatchCriteria>

`<fieldMatchCriteria>`

Contains a list of field match criteria. A rule can have zero or more field match criteria that map RTTP message fields and values. All defined field mappings must be present in the RTTP message, otherwise the rule will not match the message. Individual field mappings are defined using <match>.

Attributes: This tag has no attributes.

<group>

`<group>`

Defines a single permissioning group. A group can have zero or one <permissionSet> and zero or one <members>. Groups allow product permissions to be applied to the members of the group in an inheritance hierarchy. A user can be a member of more than one group, and groups can be members of other groups.

Attributes:

Name	Type	Default	Req?	Description
`name`	string	(none)	Y	The name of the group, which must be unique to each group. Other groups and users can become members of this group by referring to the group by this name.

<groupRef>

`<groupRef>`

Adds a group member to the group (see <group>). Groups can be members of more than one group, but cannot be members of their own or child groups.

Attributes:

Name	Type	Default	Req?	Description
`nameRef`	string	(none)	Y	The name of the group that you want to add. Only groups that have been defined using the name attribute of the <group> tag can be added to a group. Therefore nameRef must match the name attribute of a <group> tag.

<groups>

`<groups>`

Contains a list of one or more permissioning groups, with one group per child <group> tag.

Attributes: This tag has no attributes.

<master>

`<master>`

Sets the <role> of the Permissioning DataSource to master.

Attributes: This tag has no attributes.

<match>

`<match>`

A child of <fieldMatchCriteria> that defines an individual field mapping for a key/value pair. The rule will only match the RTTP message if the field identified by criteria has the value identified by value.

Attributes:

Name	Type	Default	Req?	Description
`criteria`	string	(none)	Y	The field to match.
`value`	string	(none)	Y	The value to match.

<members>

`<members>`

Defines zero or more members of a group, where each member can be a user (<userRef>) or another group (<groupRef>).

Attributes: This tag has no attributes.

<permission>

`<permission>`

Defines a single permission. A permission determines whether an action on a product will be allowed or denied. When you define a permission you can also define a namespace that will restrict the scope of the permission. If you do not define a namespace, then the permission will reside in the default namespace.

Attributes:

Name	Type	Default	Req?	Description
`action`	string	(none)	Y	The action that the permission applies to. This value should match the action defined by a matching rule (see <rule>).
`auth`	string	(none)	Y	Whether the action will be allowed or denied. Permitted values are "ALLOW", "DENY", and "NO PERMISSION" (permission neither allowed nor denied).
`namespace`	string	(none)	N	The namespace in which the permission resides. This value should match the namespace for the action defined by a matching rule (see <rule>). If not defined, the permission will reside in the default namespace.

<permissioning>

`<permissioning>`

The outermost permissioning tag, with zero or one <role>,with zero or one <rules>, zero or one <users>, and zero or one <groups>.

Attributes: This tag has no attributes.

<permissionSet>

`<permissionSet>`

Contains a list of one or more product permission sets, with one set per child <productPermissionSet> tag.

Attributes: This tag has no attributes.

<productPermissionSet>

`<productPermissionSet>`

Contains a list of one or more permissions for a set of products, with one permission per child <permission> tag.

Attributes:

Name	Type	Default	Req?	Description
`productSet`	string	(none)	Y	A comma delimited string. Each delimited section of the string must identify a single product (typically a product symbol such as "/FX/GBPUSD") or a regular expression that matches multiple products (such as ". *USD").

<role>

`<role>`

Defines the role of the Permissioning DataSource. The <role> tag must contain a <master> tag if the PermissioningDataSource is the master, or a <slave> tag if the Permissioning DataSource is a slave. If the <role> tag is omitted from the XML definition, then the PermissioningDataSource will use version 1 of the Permissioning message protocol (see Upgrading the Permissioning DataSource library 9).

Attributes: This tag has no attributes.

<rule>

`<rule>`

Defines a single permissioning rule. Every rule must define either an action attribute or an actionRef attribute, but not both.

Attributes:

Name	Type	Default	Req?	Description
`action`	string	(none)	N	The user must have permission for this action if the rule matches the RTTP message. This attribute can be used to match an RTTP message to a single action, such as "Trade" or "SPOT". If the action attribute is used then the actionRef attribute must not be used, otherwise the XML will not be valid.
`actionRef`	string	(none)	N	The name of the field in the RTTP message that identifies the action. The user must have permission for this action if the rule matches the RTTP message. This attribute can be used to match the rule when the RTTP message could define one of several alternative actions. An example would be when the value of the TradeType field could be one of SPOT, FORWARD or SWAP. If the actionRef attribute is used then the action attribute must not be used, otherwise the XML will not be valid.
`permissionNamespace`	string	(none)	N	The namespace in which the user permission for the action must reside. If a namespace is not defined, then the user must have a permission for the action in the default namespace.
`productRef`	string	(none)	Y	The name of the field in the RTTP message that identifies the product that the user must have a permission to action. The reserved value ALL_PRODUCTS means that the rule will apply to any product.
`ruleType`	string	(none)	Y	This value must always be WRITE. WRITE rules apply when data is being contributed to Liberator, and READ rules when data is being requested from Liberator. At present a default READ rule is implemented by the Permissioning Auth Module when a user attempts to view data, but in future releases of Caplin Trader it may be possible to define READ rules in XML.
`subjectNameMatch`	string	(none)	Y	The subject of the RTTP message that will match this rule. The value can be a regular expression. For example "/F." would match "/FT" and "/FI", since the "." metacharacter will match any single character.

<rules>

`<rules>`

Contains a list of one or more permissioning rules, with one rule per child <rule> tag.

Attributes: This tag has no attributes.

<slave>

`<slave>`

Sets the <role> of the PermissiongDataSource to slave.

Attributes:

Name	Type	Default	Req?	Description
`name`	string	(none)	Y	A name that uniquely identifies this slave from all other slaves of the <master>. The reserved name MASTER cannot be used to name a slave.

<subjectMapping>

`<subjectMapping>`

Maps an RTTP message subject to a subject suffix. If the user attempts to VIEW data where the subject of the RTTP message matches subjectPattern, then subjectSuffix will be appended to the subject of the RTTP message before Liberator requests the data from a DataSource. Subject mappings can be used to get pricing data from different pricing tiers, depending on the user that requested the data.

Attributes:

Name	Type	Default	Req?	Description
`subjectPattern`	string	(none)	Y	A regular expression that will be compared with the subject of the RTTP message. If a match is found, then subjectSuffix will be appended to the subject of the RTTP message.
`subjectSuffix`	string	(none)	Y	The suffix that will be appended to the subject of the RTTP message.

<user>

`<user>`

Defines a single user and the user's name and password. A user can have zero or one <permissionSet>, which allows product permissions to be applied to the user; zero or one <subjectMapping>, which allows data to be requested from a pricing tier; and zero or one <attributes>, which map user attribute names to user attribute values.

Attributes:

Name	Type	Default	Req?	Description
`name`	string	(none)	Y	The user's login name.
`password`	string	(none)	Y	The user's login password. The reserved value "keymaster" indicates that the Caplin Keymaster single sign-on system will validate the user's password.

<userAttribute>

`<userAttribute>`

Defines a single user attribute. A user attribute maps an attribute name to an attribute value.

Attributes:

Name	Type	Default	Req?	Description
`key`	string	(none)	Y	The attribute name or key.
`value`	string	(none)	Y	The attribute value.

<userRef>

`<userRef>`

Adds a user member to the group (see <group>). Users can be members of more than one group.

Attributes:

Name	Type	Default	Req?	Description
`nameRef`	string	(none)	Y	The name of the user that you want to add. Only users that have been defined using the name attribute of the <user> tag can be added to a group. Therefore nameRef must match the name attribute of a <user> tag.

<users>

`<users>`

Contains a list of one or more users, with one user per child <user> tag. Users can have product permissions applied to them.

Attributes: This tag has no attributes.

8 Further Reading

If you would like an introduction to permissioning concepts and terms or to consult reference documentation for the Permissioning DataSource API, then the following documents provide this information. You may also be interested in reading some of the other Related documents 2.

An introduction to permissioning concepts and terms

The document **Caplin Xaqua: Permissioning Overview And Concepts** introduces permissioning concepts and terms, and shows the permissioning components of the Caplin Xaqua architecture.

Reference documentation for the Permissioning DataSource API

Reference material for this API can be found in the **Permissioning DataSource: API Reference**.

9 Glossary of terms and acronyms

This section contains a glossary of terms, abbreviations, and acronyms used in this document.

Term	Definition
Action	The interaction that a user can have with a **product**.
API	Application Programming Interface
Caplin Trader	A web application framework for constructing browser-based financial trading applications (**Caplin Trader applications**).
Caplin Trader application	A **Caplin Xaqua client** that has been built using **Caplin Trader**.
Caplin Xaqua	A framework for building single-dealer platforms that enables banks to deliver multi-product trading direct to client desktops. Caplin Xaqua can also be short for a **Caplin Xaqua system**.
Caplin Xaqua client	A client desktop or web application that interfaces with **Caplin Xaqua** to deliver multi-product trading to end-users.
Caplin Xaqua system	A single-dealer platform that is built using **Caplin Xaqua**.
DataSource	An **API** and underlying code library that allows **DataSource applications** to communicate with each other.
DataSource adapter	A **DataSource application** that acts as the interface between **Caplin Xaqua** and an external (non-Caplin) system, exchanging data and/or messages with that system.
DataSource application	A **Caplin Xaqua** application that uses the **DataSource API** and code library to communicate with other Caplin Xaqua applications.
DataSource protocol	The protocol that **DataSource applications** use to communicate with each other.
Demo Permissioning DataSource	The Demo Permissioning DataSource 43 is an example of a **Permissioning DataSource** that gets its permissioning data from an XML file.
Global context	An object at the **Permissioning Auth Module**. The global context allows custom **subject mappers** to access data that is common to all subject mappers and **users**.
Group	A logical grouping of zero or more **users** and other groups, such that each group can be assigned zero or more **permissions**.
Liberator	A real-time financial internet hub that delivers trade messages and market data to and from subscribers over any network that supports web traffic.
Master	When permissioning data is sent to Liberator from multiple **Permissioning DataSource** adapters, one of the Permissioning DataSource adapters is designated the master, and the others are designated as **slaves**.
Permission	Determines whether an **action** on a **product** will be allowed or denied.
Permissioning Auth Module	One of several authentication modules that are supplied with **Caplin Xaqua**.
Permissioning DataSource	A **DataSource** adapter that acts as the interface between **Caplin Xaqua** and your **Permissioning System**.

Term	Definition
Permissioning DataSource API	The **API** that a **Permissioning DataSource** uses to send permissioning data to **Liberator**.
Permissioning System	The source of the permissioning data that you want to integrate with **Caplin Xaqua**.
Product	In permissioning documentation (including this document) a "product" is any entity on which a **User** may be assigned **permissions** (including financial instruments). In other **Caplin Trader** and **Caplin Xaqua** documentation, a "product" is a term that refers only to a financial instrument.
Role	Roles determine whether a **Permissioning DataSource** is designated as a **master** or **slave** Permissioning DataSource.
Rule	Rules link **permissions** to user interactions, and are used by **Liberator** to decide which of the many permissions that have been defined will apply when a **user** attempts to interact with a **product**.
SDK	Software Development Kit
Slave	When permissioning data is sent to Liberator from multiple **Permissioning DataSource** adapters, one of the Permissioning DataSource adapters is designated the **master**, and the others are designated as slaves.
Subject mapper	A subject mapper is a Java class that resides at the **Permissioning Auth Module**. A subject mapper can modify the message that Liberator receives when an end-user attempts to view or trade a product, and can be used to provide preferential data to selected **users**.
User	An end-user of a **Caplin Xaqua client** application such as **Caplin Trader**.

Index

Single-dealer platforms for the capital markets

CAPLIN

Contact Us

Caplin Systems Ltd

Cutlers Court

115 Houndsditch

London EC3A 7BR

Telephone: +44 20 7826 9600

www.caplin.com