CAPLIN

Caplin Xaqua 1.0

How To Create A Permissioning DataSource Adapter

November 2009

CONFIDENTIAL

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
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.

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.

Please email your feedback to documentation@caplin.com.

1.6 Acknowledgments

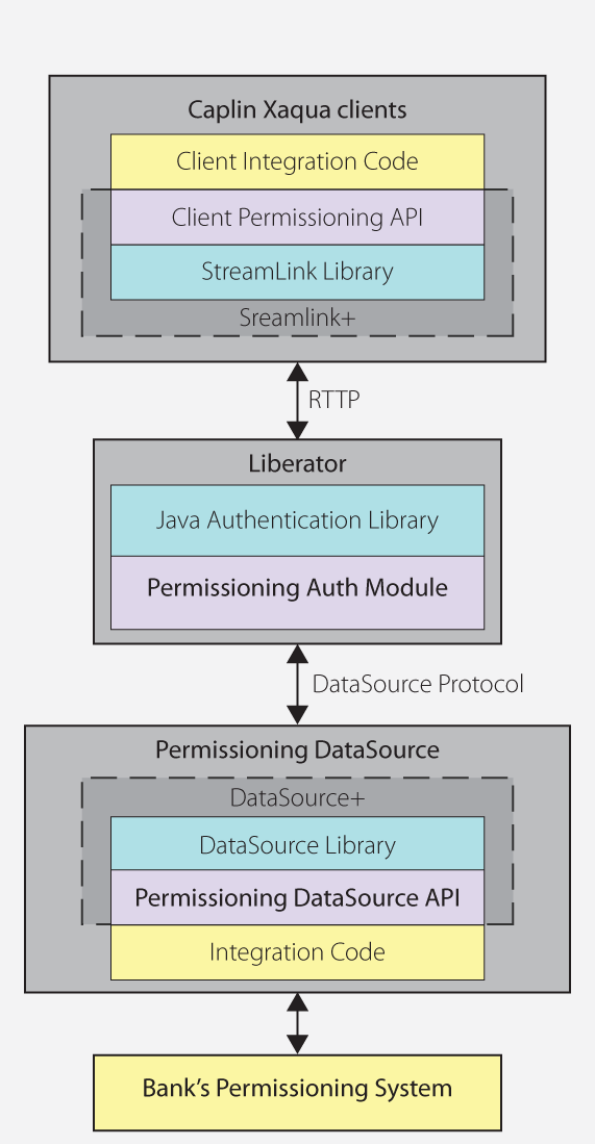
1.7 Code samples in this document

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 30. This example application uses the Permissioning DataSource API to provide Liberator with permissioning data from an XML file 33.

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.

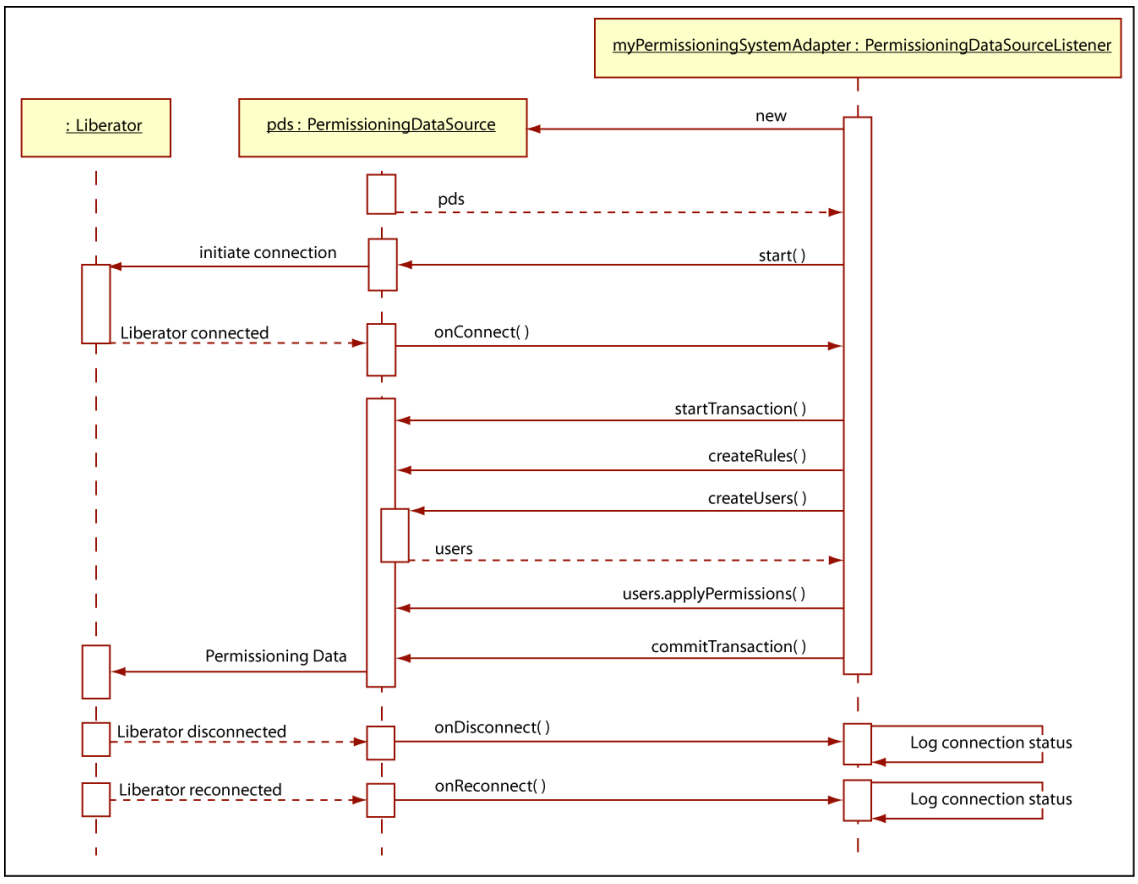
> **Tip:** For a complete description of the Permissioning DataSource API, please refer to the **Permissioning DataSource: API Reference**.

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.

3.1 Creating a Single Permissioning DataSource

The Permissioning DataSource API provides the interface between the Caplin Xaqua and a Permissioning System. When you write an application that uses this API, your code must implement the `PermissioningDataSourceListener` interface and instantiate a `PermissioningDataSource`, as summarized in the following sequence diagram and in steps 1 to 4 below.



Typical sequence of events for sending permissioning data to Liberator

1. Implement the `PermissioningDataSourceListener` interface

   This interface has three 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.

Your implementation of `onConnect()` would typically respond to this trigger by starting a transaction, applying permissioning data to the `PermissioningDataSource`, and then committing the transaction. Data is applied to the `PermissioningDataSource` when you call one of the `createUser`, `createGroup` and `create-rule` methods as part of a transaction. Committing the transaction sends that data to Liberator.

The other two callback methods, `onDisconnect()` and `onReconnect()`, simply inform your application about the Liberator connection status. There is no need to send permissioning data or start a transaction, and your implementation would typically respond by logging the connection status.

2. Instantiate a PermissioningDataSource

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

   - An instance of your `PermissioningDataSourceListener` 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 30 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 10).

4. Start the `PermissioningDataSource`

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

The following code sample is a trivial implementation of a `PermissioningDataSourceListener`, as summarized in steps 1 to 4 above.

```
// Step 1: Implement the PermissioningDataSourceListener interface
public class MyPermissioningSystemAdapter implements PermissioningDataSourceListener
{
   private PermissioningDataSource pds;

   public MyPermissioningSystemAdapter() throws IOException, SAXException
   {
      // Step 2: Instantiate a PermissioningDataSource,
      // passing 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: Start the PermissioningDataSource
      pds.start();
   }

   // Implement the onConnect() callback
   public void onConnect()
   {
      // start a PermissioningDataSource 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 );

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

   // Implement the onDisconnect() callback
   public void onDisconnect()
   {
      System.out.println("Disconnected from Liberator!")
   }

   // Implement the onReconnect() callback
   public void onReconnect()
   {
      System.out.println("Reconnected to 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 8). 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.

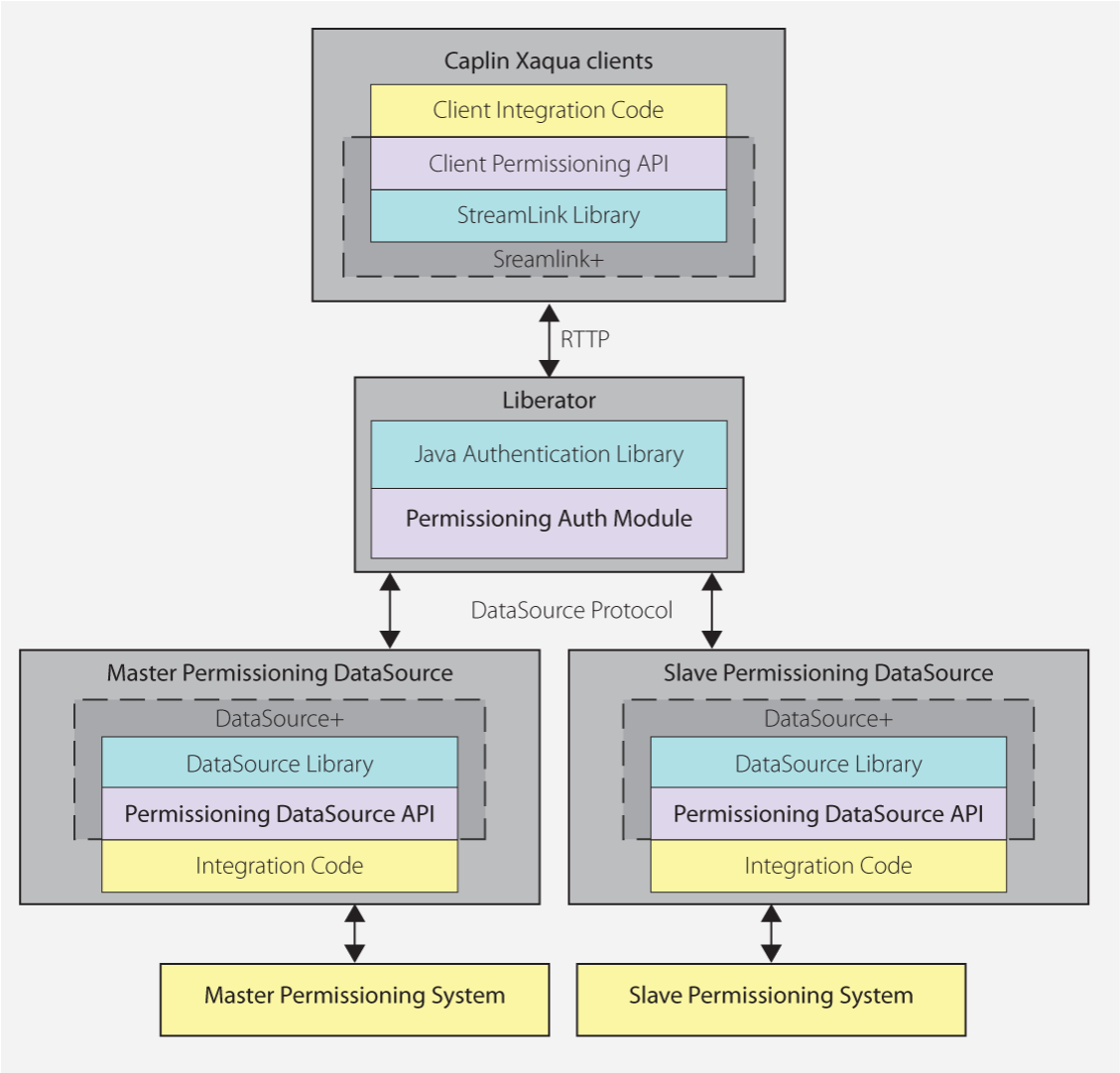
The following table shows the messaging protocols that are supported by each release of the Permissioning DataSource and Caplin Trader libraries.

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
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.



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: Implement the PermissioningDataSourceListener interface
public class MyPermissioningSystemAdapter implements PermissioningDataSourceListener
{
   private PermissioningDataSource pdsm;

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

      // Step 3: Set this PermissioningDataSource as the master
      pdsm.setMasterRole();

      // Step 4: Start the PermissioningDataSource
      pdsm.start();
   }

   // Implement the onConnect() callback
   public void onConnect()
   {
      // start a PermissioningDataSource 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 );

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

   // Implement the onDisconnect() and onReconnect() callbacks
   ...
}
```

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: Implement the PermissioningDataSourceListener interface
public class MyPermissioningSystemAdapter implements PermissioningDataSourceListener
{
   private PermissioningDataSource pdss;

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

      // Step 3: Set this PermissioningDataSource as a slave and set the name to "FX"
      pdss.setSlaveRole("FX");

      // Step 4: Start the PermissioningDataSource
      pdss.start();
   }

   // Implement the onConnect() callback
   public void onConnect()
   {
      // start a PermissioningDataSource 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( ... );

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

   // Implement the onDisconnect() callback
   public void onDisconnect()
   {
      System.out.println("Disconnected from Liberator!")
   }

   // Implement the onDisconnect() and onReconnect() callbacks
   ...
}
```

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 17).

Situation	Type of transaction required
When an `onConnect()` callback is received.	Start an image transaction. The permissioning data that you send will replace any existing permissioning data in Liberator.
When an `onReconnect()` callback is received.	This callback is for information only, you *do not* need to start a transaction or send permissioning data to Liberator.
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.

Situation	Type of transaction required
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 17).

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 17).

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 21.

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) 19.

```
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 `User.applyPermissions()` method can either be used to add a new permission to a user or to modify an existing permission.

In this example the permission to "OneClick" trade the "FX/GBPUSD" product in the "TradeType" namespace is added to the permissions already assigned to this user.

```
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 would replace any other permission the user had for this product, action and namespace.

Removing a Permission from a User

In this example we remove the permission to "OneClick" trade the "FX/GBPUSD" product that we assigned in the previous transaction (see Changing a User's Permissions 21).

```
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.

Changing a Group's Permissions

The `Group.applyPermissions()` method can either be used to add a new permission to a group or to modify an existing permission.

In this example we remove the permission to trade all FX products, and add a permission to trade a small set of FX products.

```
pds.startUpdateTransaction();

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

// remove the promiscuous permission for all FX products
Set oldProducts = new HashSet();
oldProducts.add("/FX/.*");
group.removePermission(oldproducts, "TradeType", "OneClick");

// allow "OneClick" action on a small, explicit set of FX products
Set newProducts = new HashSet();
newProducts.add("/FX/GBPUSD");
newProducts.add("/FX/GBPAUD");
group.applyPermission(newProducts, "TradeType", "OneClick", Authorization.ALLOW);

pds.commitTransaction();
```

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

Changing the Subject Mapping for a User

A default subject mapper is provided with the Permissioning software that allows one subject to be mapped for each user.

If you want to provide multiple subject mappings for a user, or if you want to provide customized mapping logic, then you must create a custom subject mapper.

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.

Using a custom subject mapper

If you want to map subjects using a custom subject mapper, then the set`SubjectMapper()` method specifies the class that implements the subject mapper, and the `addSubjectMapping()` method adds the subject mappings.

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()` maps subjects when you are using a custom subject mapper, but `setSubjectMapping()` maps subjects 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).

To create your own subject mapper, see Creating a Custom Subject Mapper 25.

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 to be mapped for each user. If you want to map more than one subject for a user, or if you want to provide 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 25 of the Permissioning DataSource API.
- Compile the custom Java code and deploy it to the Permissioning Auth Module 29, and configure Liberator to use the compiled code.

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 provides two 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 user.

  The `subject` passed to this method is the RTTP message received by Liberator, and must be parsed to determine whether or not a key is present in the RTTP message.

  If a mapping exists for the subject (and any identified key), then return the modified RTTP message to Liberator as a string. Liberator will use the modified RTTP message to communicate with the DataSource and to check user permissions.

  If a mapping does not exist for the subject, then return null. In this case Liberator will use the original RTTP message to communicate with the DataSource and to check user permissions.

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.

```
import java.util.HashMap;
import java.util.Map;
import com.caplin.permissioning.SubjectMapper;

public class MyCustomSubjectMapper implements SubjectMapper
{
  // The subject passed to mapSubject() could contain these parameters
  public static final String DEFAULT_MAPPING = "DEFAULT";
  public static final String TRIGGER_PARAM = "TRIGGER_PARAM";

  // Set up a Map that will store the keys and subject mappings passed
  // to updateMappings()
  private final Map<String,Map<String,String>> mappings =
          new HashMap<String,Map<String,String>>();

  // Do not provide a constructor, but allow the compiler to create the default
  // (no-arg) constructor (so that instances of the custom SubjectMapper class
  //can be created dynamically)

  // implement updateMappings()
  // this method is called by the Permissioning Auth Module
  // when new key and subject mappings received from the Permissioning DataSource
  public void updateMappings(String key, Map<String, String> updateMap)
  {
    Map<String,String> keyedMap = mappings.get(key);
    // add the key and subject mappings if the key does not exist
    if(keyedMap == null)
    {
      mappings.put(key,new HashMap<String,String>(updateMap));
    }
    // add subject mappings for this key if the key does exist
    else
    {
      keyedMap.putAll(updateMap);
    }
  }

  // implement mapSubject()
  // called by the Permissioning Auth Module when RTTP message received from a user
  public String mapSubject( String subject )
  {
    // parse the passed in subject.
    // The subject is a String containing the subject body and a list of parameters
    ParsedSubject parsedSubject = new ParsedSubject(subject);
    Map<String,String> params = parsedSubject.getParameters();
    // does a parameter identify a key
    String triggerParam = params.get(TRIGGER_PARAM);
    // return null if no key present in the subject
    if(triggerParam == null)
    {
      return null;
    }
```

```
  // if there is a key, set mapping to the key value
  Map<String,String> mapping = mappings.get(triggerParam);
  // look for the key mapping and subject mapping in the store
  return findTier(mapping, parsedSubject.getSubject());
}

// search the store for the key mapping and subject mapping
private String findTier(Map<String,String> mapping, String subject)
{
// return null if the key is not in the store
  if(mapping == null)
  {
    return null;
  }
  // if the key and subject mapping exists in the store, return concatenated
  // subject pattern and subject suffix from store
  String mappedTier = mapping.get(subject);
  if(mappedTier != null)
  {
    return subject + mappedTier;
  }
  // if the key mapping exists in the store but not subject mapping, return
  // concatenated subject pattern and default subject mapping
  String defaultTier = mapping.get(DEFAULT_MAPPING);
  if(defaultTier != null)
  {
    return subject + defaultTier;
  }
  return null;
}

// Class that parses the subject passed to mapSubject(String subject),
// where subject is of the form:
// "subjectBody;param1=value, param2=nalue, ... , paramN=value"
// Instantiated class consists of the subjectBody and a Map of parameters
// (param1 to paramN)
private static class ParsedSubject
{
  private final Map<String,String> params;
  private final String fullSubject;
  private final String subject;
  private static final String PARAM_START_CHAR = ";";
  private static final String PARAM_SEPERATOR_CHAR = ",";

  public ParsedSubject(String fullSubject)
  {
    this.fullSubject = fullSubject;
    int paramStart = this.fullSubject.indexOf(PARAM_START_CHAR);
    this.subject = this.fullSubject.substring(0, paramStart);
    this.params =parseParams(this.fullSubject.substring(paramStart+1));
  }

  private Map<String,String> parseParams(String paramString)
  {
    Map<String,String> paramMap = new HashMap<String,String>();
    if(paramString == null || paramString.length() == 0)
    {
      return paramMap;
    }
    String[] split = paramString.split(PARAM_SEPERATOR_CHAR);
    for(String element : split)
    {
      String[] keyValue = element.split("=");
      paramMap.put(keyValue[0],(keyValue.length > 1)? keyValue[1]: "");
    }
    return paramMap;
  }
```

```
    // return subjectBody
    public String getSubject()
    {
      return subject;
    }

    // return map of parameters (param1 to paramN)
    public Map<String,String> getParameters()
    {
      return params;
    }
  }
}
```

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

At the Permissioning DataSource

The Permissioning DataSource sends a subject mapping for the key="FX". The subject mapping maps "EUDUSD" to "-tier1". The key and subject mappings are sent as part of a transaction to the Permissioning Auth Module using the methods `User.setSubjectMapper()` and `User.addSubjectMappings()`.

At the Permissioning Auth Module

When the transaction is received from the Permissioning DataSource, the Permissioning Auth Module calls the interface method `updateMappings()`, passing in the key and subject mapping. The `updateMappings()` method creates a map of the key ("FX") and subject mapping ("EUDUSD" to "-tier1"), and stores this mapping.

When the user requests a price for an FX instrument, the Permissioning Auth Module calls the interface method `mapSubject()`, passing in the subject (the RTTP message) that requested the price. In this example the subject takes the from "`subjectBody;param1, param2, ... paramN`", where one of the parameters (`param1` to `paramN`) identifies the "FX" key.

The `mapSubject()` method parses the passed in subject, finds the "FX" key in the parsed parameters, and looks for the matching "FX" key in the map of stored keys. Mapped against this stored key is the subject mapping "EURUSD" to "-tier1". If `subjectBody` is "EURUSD", then `mapSubject()` returns the concatenated string "EURDUSD-tier1". If `subjectBody` is not "EURUSD", then `mapSubject()` returns null.

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

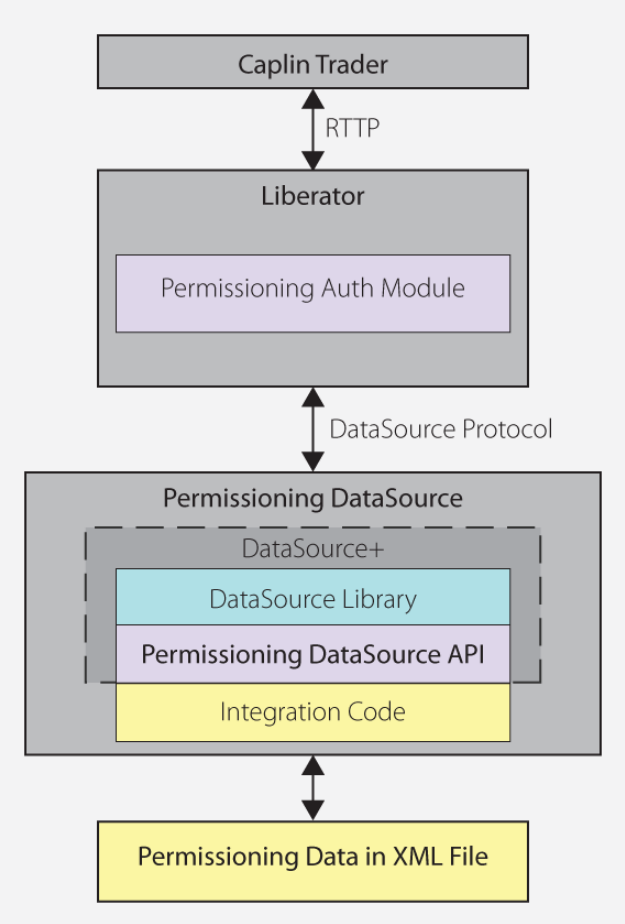
JAR files are typically copied directly to */lib/java* in the Liberator installation directory. When you have copied the JAR file, add the classpath for the JAR file to the Liberator configuration file *java.conf*.

```
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 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).

5.1 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.

5.2 Overview of the Demo Permissioning DataSource

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

`PermissionsLoader`: This interface defines a service that will load 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 `PermissioningDataSourceListener` 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.

  Starts the `DemoPermissioningDataSource`.
- `start()` Retrieves permissioning data from the permissioning system and initiates a connection to Liberator.
- `onConnect()` Called by the `PermissioningDataSource` when a Liberator connection is established.
- `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 `PermissioningDataSourceListener` interface and `PermissioningDataSource` class are described in the **Permissioning DataSource: API Reference**.

6 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.

6.1 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.

6.2 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 36 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)

6.3 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 10).

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.

7 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**.

8 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 **Caplin Xaqua client** application written in Ajax that provides a framework and comprehensive set of components for constructing browser-based trading applications. Caplin Trader was formerly called "Caplin Trader Client".
Caplin Xaqua	A single-dealer platform that enables banks to deliver multi-product trading direct to client desktops. Caplin Xaqua was formerly called "the Caplin Platform".
Caplin Xaqua client	A client desktop application that interfaces with **Caplin Xaqua** to deliver multi-product trading to end users. The application can be implemented in any technology that is supported by Caplin Xaqua; for example Ajax, Microsoft .NET, Microsoft Silverlight™, Adobe Flex™, and Java™.
DataSource	DataSources are software adapters within **Caplin Xaqua** that connect Xaqua to external sources of real time data and external **Permissioning Systems**. In other Caplin documents DataSources are also called DataSource adapters.
Demo Permissioning DataSource	The Demo Permissioning DataSource 30 is an example of a **Permissioning DataSource** application that gets its permissioning data from an XML file.
Group	A logical grouping of zero or more **users** and other groups, such that each group can be assigned zero or more **permissions**.
Liberator	Caplin Liberator is a bidirectional streaming push server designed to deliver market data and trade messages 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**.
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

Term	Definition
	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.
User	An end user of a **Caplin Xaqua client** application such as **Caplin Trader**.

Index

- S -

- T -

- U -

Contact Us

Caplin Systems Ltd
Triton Court
14 Finsbury Square
London EC2A 1BR
Telephone: +44 20 7826 9600
Fax: +44 20 7826 9610
www.caplin.com