Sun Java™ System RFID Software 3.0 Developer's Guide

Sun Microsystems, Inc.
www.sun.com

Part No. 819-4686-10
February 2006, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Adobe PostScript

Contents

Before You Begin

This developer's guide for Sun Java™ System RFID Software 3.0 (RFID software) contains information for the Enterprise software developer who needs to access the data from the RFID tag reader system. The Developer's Guide is not aimed at the reader adapter developer. Reader adapter information is covered in the *Sun Java System RFID Software Toolkit Guide* included with the Sun Java System RFID Software Toolkit. The adapter information is in the `AdapterDevelopment.pdf` file.

Screen shots vary slightly from one platform to another. Although almost all procedures use the interface of the RFID software components, occasionally you might be instructed to enter a command at the command line.

Before You Read This Book

You should be familiar with RFID concepts and with the following topics:

- Jini™ network technology concepts
- Java™ programming and concepts
- Java™ DataBase Connectivity technology- JDBC™ concepts and usage
- Java™ 2 Platform, Enterprise Edition (J2EE™) technology and usage
- Client-server programming model
- Familiarity in managing large enterprise systems
- Administration of one of the supported application servers
- Administration of one of the supported databases

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content goods or services that are available on or through such sites or resources.

Documentation Formatting Conventions

This section describes the types of conventions used throughout this guide:

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

- File and directory paths are given in UNIX® format (with forward slashes separating directory names).
- URLs are given in the format:

  `http://`*`server.domain`*`/`*`path`*`/`*`file`*`.html` where *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename.
- UNIX-specific descriptions throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.cvspass` file. Use `DIR` to list all files. `Search is complete.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`> login` `:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* save your changes.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type **`DEL`** *filename*.

Related Documentation

The following table lists the tasks and concepts that are described in the Sun Java™ System RFID Software manuals and *Release Notes*. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate manual.

For information about	See the following
Late-breaking information about the software and the documentation	*Sun Java System RFID Software 3.0 Release Notes*
Installing Sun Java™ System RFID Software	*Sun Java System RFID Software 3.0 Installation Guide*

For information about	See the following
The following RFID Software administration topics: • RFID Software overview • Configuring the RFID Event Manager • Configuring Communication with SAP AII • Using the RFID Management Console • Configuring the RFID Information Server • RFID device adapter reference • RFID Event Manager component reference • RFID Event Manager configuration file reference	*Sun Java System RFID Software 3.0 Administration Guide*
The following topics for RFID software developers: • Introduction to Sun Java System RFID Software programming platform • Creating custom filters and connectors • Using RFID Device client APIs • Using web services for device access • Using Application Level Event (ALE) web services API • Using RFID Information Server client APIs • PML utilities	*Sun Java System RFID Software 3.0 Developer's Guide*

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address: `docfeedback@sun.com`

Please include the part number (819-4686-10) of the document in the subject line of your email.

CHAPTER 1

Introduction to Sun Java System RFID Software Programming Platform

This chapter describes the architecture of the Sun Java™ System RFID Software 3.0 (RFID Software) and introduces the programming mechanisms that are available for using the information generated by the physical devices that comprise your RFID network. Subsequent chapters contain more details. The following topics are covered in this chapter:

RFID Software Architecture Overview

The RFID Software consists of the following four major modules:

- RFID Event Manager
- RFID Configuration Manager (a component of the RFID Event Manager)
- RFID Management Console
- RFID Information Server

The RFID Event Manager communicates with RFID sensor devices to gather information from the physical world. This information can be stored in the RFID Information Server for future analysis. The information can also be sent continuously to third-party applications as it arrives at the RFID Event Manager.

The RFID Configuration Manager is a graphical user interface (GUI) application that is used to specify the set of devices connected to the RFID Event Manager. You also use the RFID Configuration Manager to statically define how to process the information within the RFID Event Manager and where to send the information after this processing.

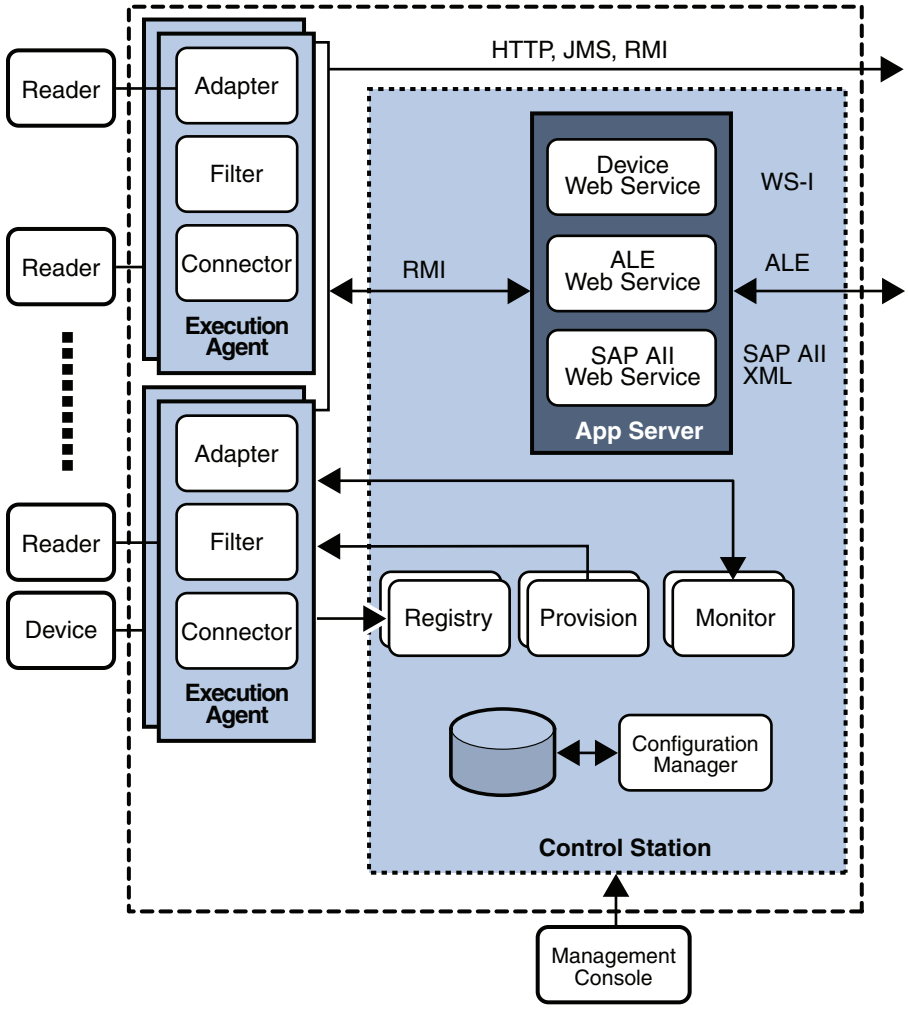
You can also use the RFID Event Manager to dynamically specify how to process the incoming information and define the subsequent consumers of the processed information. This guide describes the programming mechanisms that developers can use to dynamically control sensor devices and to define rules for processing the information collected by the devices.

You use the RFID Management Console to monitor and manage the status of the devices that are connected to the RFID Event Manager. The RFID Management Console enables system administrators to monitor statistics and change runtime parameters for each of these devices.

The RFID Event Manager is a distributed platform consisting of a single Control Station and one or more Execution Agents. In the simplest and most common scenario, the Execution Agent and Control Station are installed on the same computer. The Execution Agent is responsible for communicating with the physical devices, processing the information, and posting the information to the *consumers* of the information. The system administrator uses the RFID Configuration Manager, which is installed as part of the Control Station component, to create one or more Configuration Objects. A Configuration Object specifies one or more devices to control and specifies a set of components that process the device information. The set of information-processing components is called a Business Processing Semantic Unit (BPS).

In the simplest scenario, each Configuration Object is executed by a single Execution Agent. In a large RFID network with many deployed devices, the RFID Event Manager functionality is scaled by installing multiple Execution Agents on separate computers. In this more complex deployment, the Control Station provisions each of the Configuration Objects to a separate Execution Agent in round-robin fashion. The Control Station continuously monitors the status of the Execution Agents. To provide high processing availability, if an Execution Agent fails, the Configuration Object is provisioned to another Execution Agent.

The following illustration shows the overall components and communication flows comprising the RFID Software.


HTTP, JMS, RMI
Reader
Adapter
Filter
Connector
Reader
Execution Agent
RMI
Device Web Service
WS-I
ALE Web Service
ALE
SAP AII Web Service
SAP AII XML
App Server
Adapter
Reader
Filter
Device
Connector
Execution Agent
Registry
Provision
Monitor
Configuration Manager
Control Station
Management Console

Structure of a Configuration Object

A Configuration Object consists of a collection of components called adapters, filters, and connectors. The components are linked in a chain to process events arriving from the RFID devices. Typically, an adapter component begins the chain by receiving information from the device and sending information to the device. Filters are used in the middle of the chain to remove *noise*, such as excessive read events, or to perform other data manipulation. Connectors (also called *loggers*) are at the end of the chain to collect data for back-end processes or *listeners* (also called consumers). A back-end process might be the RFID Information Server or a third-party software application. The following figure shows an example of a Configuration Object receiving information from an RFID reader and posting the processed information to the RFID Information Server.

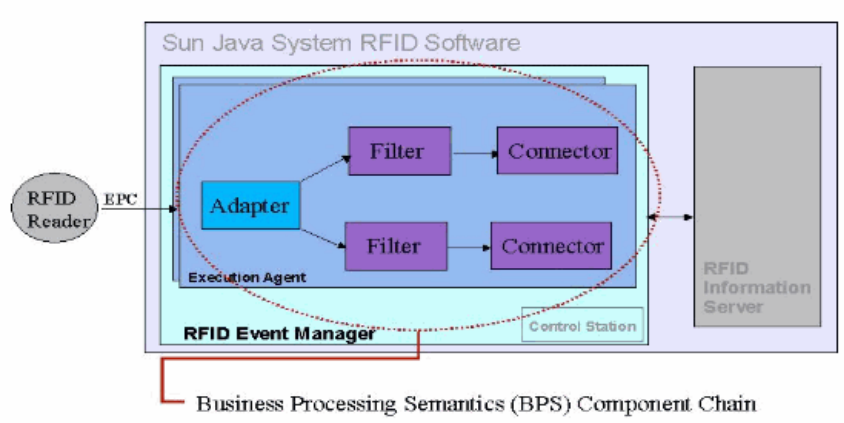


FIGURE 1-1 Structure of the RFID Event Manager Configuration Object.

The components of the Configuration Object are defined as follows:

- **Adapter** – This component implements the communication protocol for a specific type of device. The adapter is equivalent to an operating system driver. The adapter handles the details of communicating with the hardware devices. Each adapter collects and transmits information to a specific manufacturer and model of RFID device. The adapter then transmits the collected information to the next component in the BPS.
- **Filter** – A filter component might modify the information flowing from an adapter to a connector in the following ways:
  - The filter can reduce the amount of information in a meaningful way. For example, a filter can prevent duplicate events from passing through to the connector.

- The filter can add metadata to the information going to the connector. This metadata can add meaning to the event. For example, you might locate and configure multiple readers in a doorway. Then you might use a filter to indicate to the connector when an event represents a tag coming in or going out of the doorway. Using this method reduces the quantity of data needing to be processed by the RFID application. By sending only clean, complete information to the application, the application code can focus on business-oriented functionality, rather than on sifting through raw, unprocessed event data.

- **Connector** – A connector sends the information originating from the RFID readers to applications that use the information. Connectors *bind* the RFID Event Manager to RFID or sensor-based applications. A connector puts the information into an acceptable format for the application. Many of the connectors provided with the RFID Software 3.0 release package the data in XML messages that conform to the Auto-ID Center PMLCore specification. See Chapter 7 for details on using the Java library included in the RFID Event Manager installation.

Note – These RFID Event Manager components are subclasses of `com.sun.autoid.ems.AbstractComponent`, a class that follows the Composite design pattern (described in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides). This pattern enables the RFID Event Manager to work with filters, connectors, and adapters using the same interface at a basic level. The `AbstractComponent` class also provides various services required by these three class types.

RFID Event Processing Basics

As the name implies, the RFID Event Manager is an event-driven platform. Components (adapters, filters, and connectors) generate events to communicate with each other. Every component is an event producer, and every component can register itself with other components to consume their events. Typically, an adapter listens and/or queries the physical device for information. The information is then packaged inside an event object, and this event object is posted to the listeners that were previously registered with the adapter. As shown in FIGURE 1-1, the adapter receives Electronic Product Code (EPC) information from the RFID device, packages the information as an `Identifier` object and posts the event to the two filters that have previously registered themselves as consumers of the adapter events. The filters, in turn, manipulate the information and post the events to the respective

connector, which consumes the events from a particular reader. The connector then packages the event in a manner that is understood by the application that consumes the event.

Note – The Javadoc documentation for the RFID Software is available as part of the Sun Java System RFID Software Toolkit. See "To Download and Install the RFID Software Toolkit" on page 22 for information on getting the toolkit.

`Identifier` Objects

RFID data captured by adapters is packaged in `Identifier` objects. See the API specifications (Javadoc documentation) for the `com.sun.autoid.identity` package – *rfid-toolkit-dir*`/docs/api/index.html`.

`Identifier` Objects include the following:

- Numeric identifier – A general purpose `Identifier` object consisting of a number.
- EPC identifier – The Electronic Product Code (EPC) assigned to a physical item. Several subcategories are defined as follows:
  - EPC_GIAI – Global individual asset identifier
  - EPC_GID – General identifier
  - EPC_GRAI – Global reusable asset identifier
  - EPC_SGLN – Serialized global location number
  - EPC_SGTIN – Serialized global trade identification number
  - EPC_SSCC – Serialized shipping container code
  - DoD – U. S. Department of Defense construct

In addition to the unique ID, the `Identifier` object might contain a set of generic properties. The properties can be interpreted by the filters in an application-dependent manner. See the Javadoc documentation for the `RfidTag` object.

`Event` Objects

The RFID Event Manager components, adapters, filters, and connectors, can generate and consume `event` objects from other components. See the API specifications for the `com.sun.autoid.event` package – *rfid-toolkit-dir*`/docs/api/index.html`. The base class is `Event`.

`Event` objects include the following:

- `Identifier` events as follows:

a. `IdentifierEvent` – This object carries a single `Identifier` object, which represents the EPC (Electronic Product Code) or other type of identifier detected by the device, and any other properties that are specific to the adapter and application.

b. `IdentifierListEvent` – This object is similar to `IdentifierEvent`, but carries multiple `Identifier` objects that were detected by the same device.

c. `DeltaEvent` – This object carries multiple `Identifier` objects, but separates the information into two categories:

i. Objects coming into the device's field of view

ii. Objects moving out of the device's field of view

- `MiscEvent` – The `MiscEvent` object carries an event containing a generic set of properties defined by the adapter or filter implementer. The properties are application dependent.
- `StatusEvent` – The `StatusEvent` object carries a message describing the status of the device.

This guide provides details on how to create custom filters and connectors. You can create custom event types, provided they are sub-classed from `com.sun.autoid.event.Event`. If you create your own event types, then your filter cannot feed other filters and connectors that do not know how to handle the new event type. The best practice is to support the `IdentifierEvent`, `IdentifierListEvent`, `DeltaEvent`, `StatusEvent`, and `MiscEvent` types in your filters and connectors and refrain from creating new event types unless absolutely necessary.

Processing RFID Event Manager Information

There are four main mechanisms for consuming information generated by the devices connected to the RFID Event Manager.

- **RFID Configuration Manager** – Use the RFID Configuration Manager to statically define one or more connectors that post information to an application. This requires the least amount of programming by an application developer, but is restricted to one-way notifications initiated by the RFID Event Manager. See the *Sun Java System RFID Software 3.0 Administration Guide* for details on using the RFID Configuration Manager.

- **EPCglobal Application Level Events (ALE)** – Use ALE to specify the type of events the application consumes. When using ALE, the application generates XML messages that define the events of interest. The application needs to be programmed to handle the XML messages that are received containing the requested information. To facilitate development, the Sun Java System RFID Software Toolkit provides a Java library implementing the necessary APIs to use ALE. See Chapter 5 of this guide for more details.

  The EPCglobal *Application Level Events (ALE) Specification, Version 1.0* specifies only one-way communication to the device to obtain RFID tag identifier information. The specification does not implement a mechanism for getting tag user data, for programming tags, or for managing the devices. Any Java or non-Java application that complies with the EPCglobal ALE 1.0 specification can communicate with the RFID Event Manager.

  At the time of this release, EPCglobal is in the process of defining a new version of the ALE specification that will specify a standard method to obtain user data and program tags, among other things. Sun Microsystems, Inc. is actively participating in the definition of this standard and plans to provide an updated library and implementation as the specification process progresses.

- **Java APIs** – Use the Java `ReaderClient` APIs to control devices, program RFID tags, read and write user memory and tag identification on RFID transponders (tags) using the Java library bundled with the RFID Software 3.0. Your application communicates directly with the RFID Event Manager by using Java RMI (Java Remote Method Invocation) without the need to convert between protocols and data representation. See Chapter 3 of this guide for more details.

- **Web services for device access** – Use the web services for device access when you need to cross firewalls using SOAP or you need to communicate with the RFID Event Manager from a non-Java application. This mechanism gives you the same functionality as the Java `ReaderClient` API, but is implemented as a web service. The RFID Event Manager and the client application exchange SOAP messages when using this mechanism. SeeChapter 4 of this guide for more details.

These mechanisms are not mutually exclusive. You can use them in conjunction with each other to achieve a task. You might choose to use ALE to obtain tag information, and then use the device access web services to get user data and program tags. There may be applications where you want to mix and match web service and Java APIs. For example, you might use the Java `ReaderClient` API to communicate with RFID devices in your local network and use the ALE Java library or the device access web services to communicate with an RFID Event Manager in a remote network across the continent.

Managing RFID Event Manager Devices

Many times an application needs to get device identification or to control the gathering and sending of information. Use the Java `ReaderClient` APIs or web services for device access APIs to instruct a device when to start gathering data, to stop gathering data, to get device status and to get device identification information.

System administrators can use the RFID Management Console to perform these same tasks from a web interface without any need for programming. See the *Sun Java System RFID Software 3.0 Administration Guide* for details on using the RFID Management Console.

CHAPTER 2

Creating Custom Filters and Connectors

This chapter describes how to create your own custom filters and connectors. The Sun Java System RFID Software Toolkit includes a sample template for creating a filter. The procedures in this chapter use the RFID Software Toolkit in conjunction with the NetBeans™ IDE, version 4.1 to describe the creation of a sample custom filter. The following topics are included:

Setting Up Your NetBeans Environment

The Sun Java System RFID Software Toolkit includes a NetBeans plug-in (.nbm), `com-sun-autoid-toolkit.nbm`, that contains various templates and code examples. One of these templates is the EPCTypeFilter. You can us it to simplify the creation of custom components for the RFID Software. The `RFID Software Toolkit .nbm` contains source templates to use as a basis for your custom code.

To install the RFID Software Toolkit NetBeans plug-in, you first download and install the NetBeans 4.1 IDE. Then, you need to set up your development environment using the following procedures:

▼ To Download and Install NetBeans

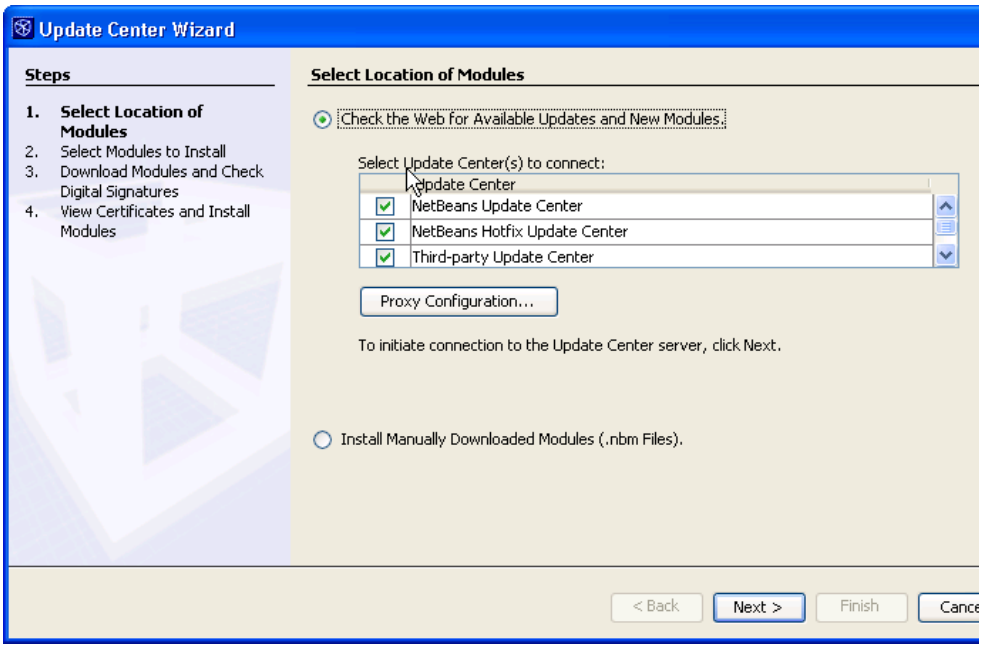
1. **Download NetBeans 4.1 from** `http://java.sun.com/j2se/1.4.2/download-netbeans.html`**.**
2. **Install the NetBeans IDE according to the instructions.**
3. **Use the following procedures to set up the NetBeans environment for your RFID component development.**

▼ To Download and Install the RFID Software Toolkit

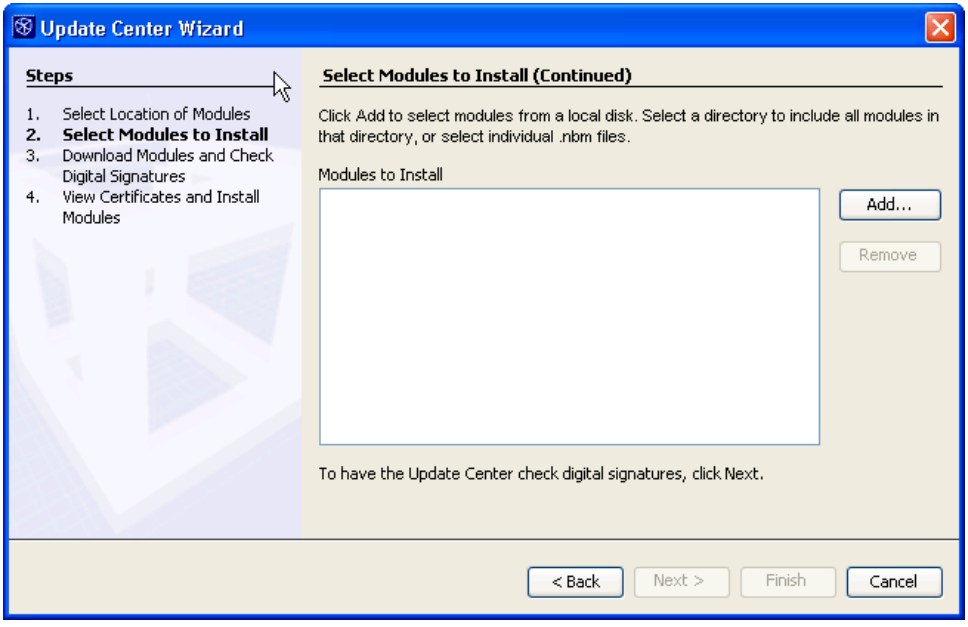
1. **Download the RFID Software Toolkit,** `RfidToolkit.zip`**, from the Sun Partner Advantage web site** `http://partneradvantage.sun.com/`**.**

   The software toolkit is available at the following URL: `http://partneradvantage.sun.com/protected/partners/technology/rfid/` once you log in to the membership center.
2. **Unzip the file to a directory on your system.**
3. **After launching NetBeans, choose Tools → Update Center.**

   The Update Center Wizard appears.

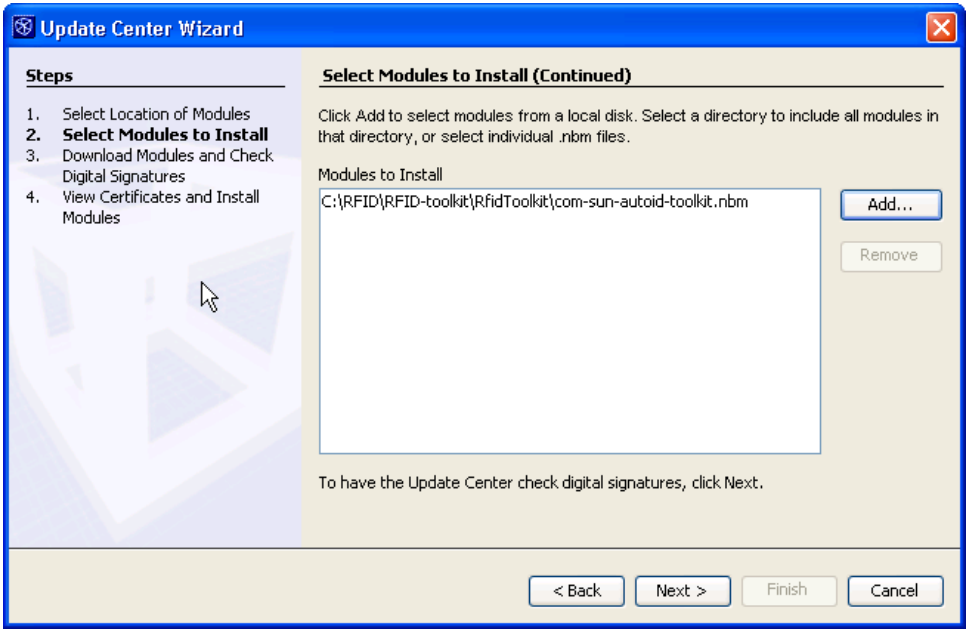


4. **Select Install Manually Downloaded Modules (.nbm files) and click Next.**



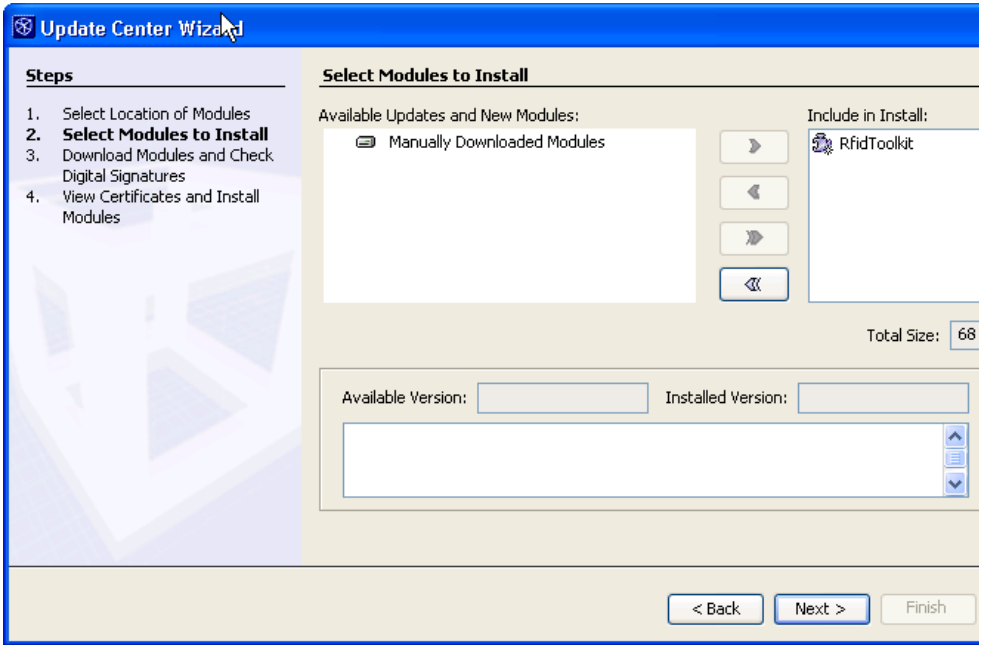
5. **Click Add and browse to the `com-sun-autoid-toolkit.nbm` file from the unzipped `RfidToolkit.zip` archive.**

6. **Select the NBM and click OK.**



7. **Click Next.**

   The RfidToolkit appears in the Include in Install list.



8. **Click Next and accept the license agreement by clicking Accept.**

   You see a short visual cue that the IDE updater is installing the module.

9. **When the installation is complete, click Next.**

   The View Certificates and Install Modules page appears.

10. **Select the check box next to RfidToolkit - version 3.**

    An Unsigned Module dialog box appears.

11. **Select Yes and click Finish.**

    Now you are ready to set up your samples. To use the RFID filter project, proceed to the next section.

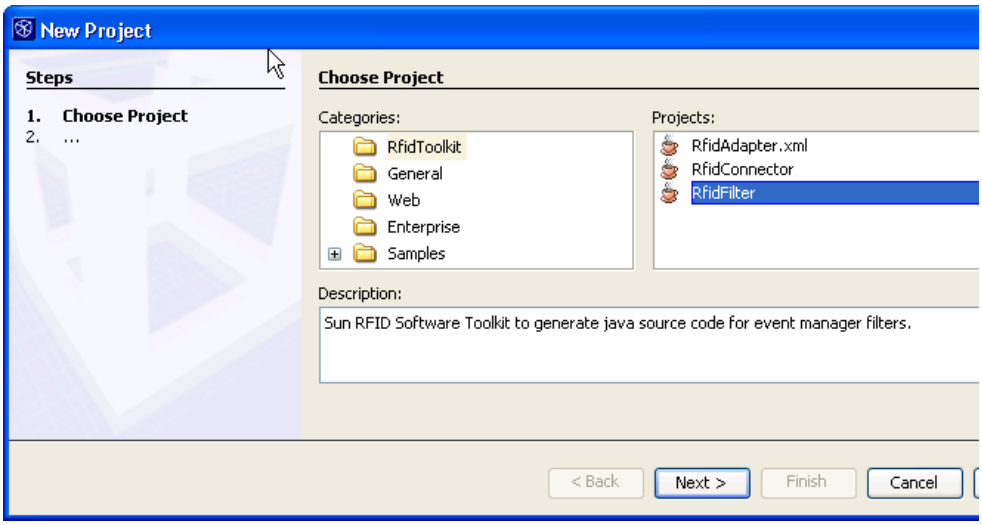
▼ To Set Up the Example Filter Project

1. **In NetBeans, choose File → New Project.**

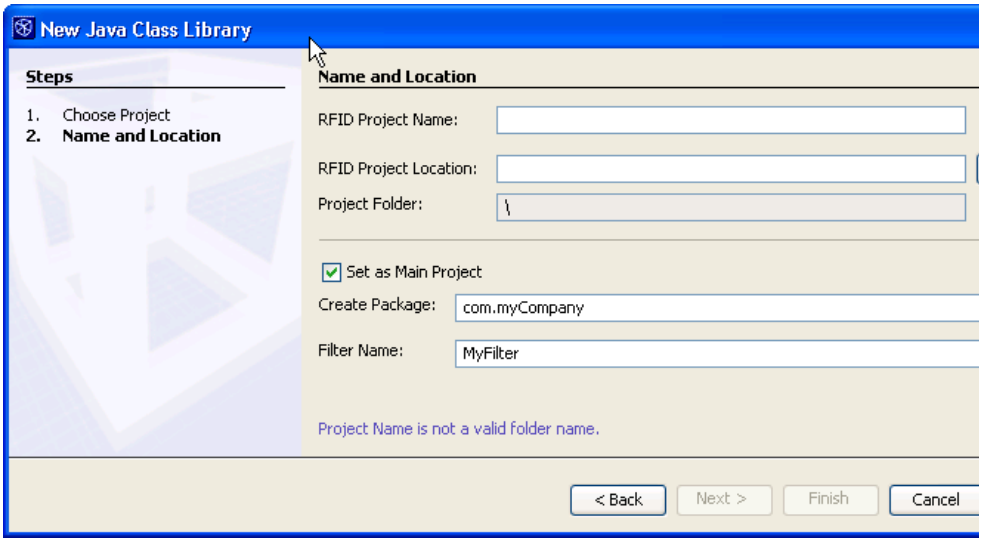
   The New Project wizard appears.

2. **Under Categories, select RfidToolkit.**

3. **Under Projects, select RfidFilter.**



4. **Click Next.**



5. **In the RFID Project Name field, type `EPCTypeFilter`.**

   Notice that when you type the RFID Project Name, the IDE automatically suggests this name for the name of the RFID Project Folder.

6. **For the RFID Project Location field, click Browse, navigate to any directory on your computer and create a new folder called `Projects`.**

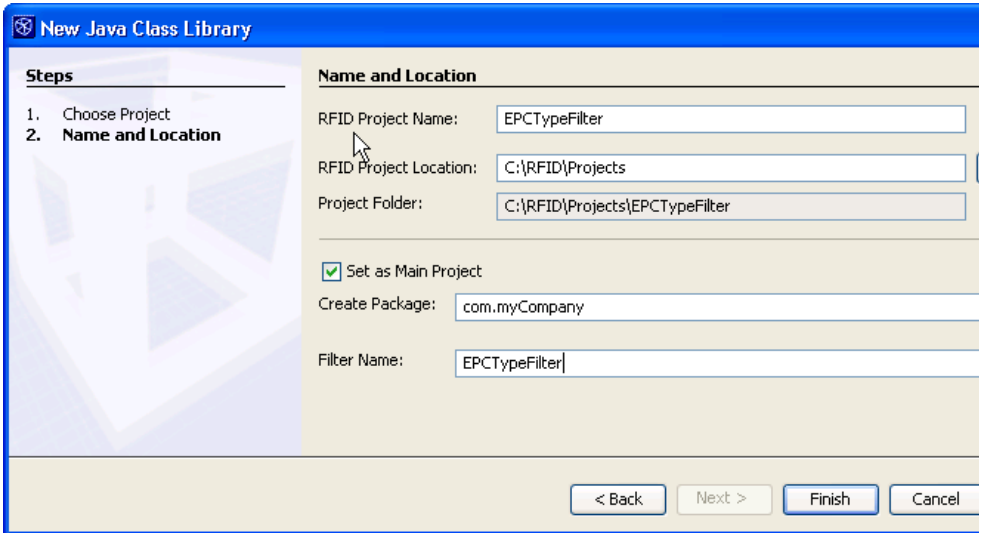
   Your RFID projects will be stored in this location.

7. **Confirm that Set as Main Project is selected.**

8. **In the Create Package field, type the value,** `com.mycompany`**.**

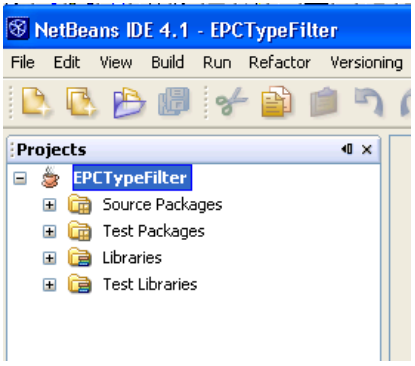
   This is the package name of the filter.

9. **In the Filter Name field, type** `EPCTypeFilter`**.**



10. **Click Finish.**

    Your RFID filter project appears in the Projects window.



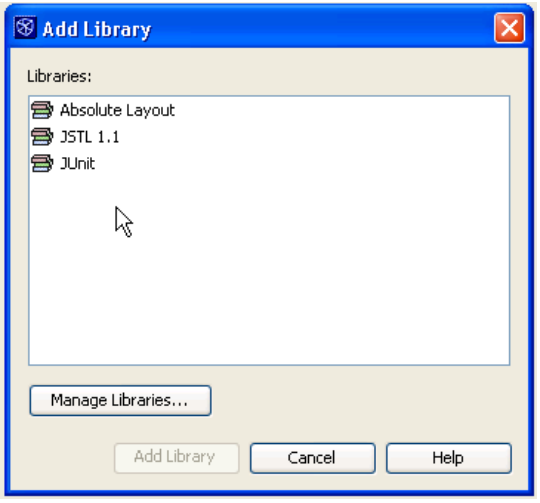
Note – To use the built-in JUnit tests, you need to add `JAR` files to the RFID library. Continue to the next procedure.

▼ To Create the RFID Library for the Custom Component Examples

Use this procedure to create a library that is used for both the RFID filter and connector projects.

1. **In the IDE Projects window, under the EPCTypeFilter node, right-click Libraries.**
2. **Select Add Library from the context menu.**

   A dialog appears. You need to create a new library called RFID.



Note – Adding a new RFID library only has to be done once. Use this procedure for the first RFID custom component project. After the library has been added, you can go directly from step 5 to step 11.

---

3. **Click Manage Libraries in the dialog.**
4. **Click New Library and type the following values:**
   - Library Name: RFID
   - Library Type: Class Libraries
5. **Click OK.**
6. **Under Libraries, confirm that the RFID library is selected.**
7. **Now add the required JAR files as listed for your platform by performing these steps for *each* of the JAR files to be added:**

   a. **Click Add Jar/Folder on the right hand side of the dialog.**

b. Browse to the JAR file location and select the JAR file.

c. Click Add Jar/Folder to add the JAR file to the library.

The following tables show the location of the JAR files for each supported platform.

TABLE 2-1 Location of JAR Files on Solaris

Location	Name of Required JAR file
/opt/SUNWrfid/lib	sun-rfid-common.jar
/opt/SUNWrfid/lib/util	concurrent.jar
/usr/share/lib	jaxb-api.jar jaxb-impl.jar xsdlib.jar jaxb-libs.jar jax-qname.jar namespace.jar relaxngDatatype.jar
/opt/SUNWjdmk/5.1/lib	jmx.jar

TABLE 2-2 Location of JAR Files on Linux

Location	Name of Required JAR file
/opt/sun/rfidem/lib	sun-rfid-common.jar
/opt/sun/rfidem/lib/utils	concurrent.jar jaxb-api.jar jaxb-impl.jar xsdlib.jar jaxb-libs.jar jax-qname.jar namespace.jar relaxngDatatype.jar
/opt/sun/jdmk/5.1/lib	jmx.jar

TABLE 2-3 Location of JAR Files on Windows

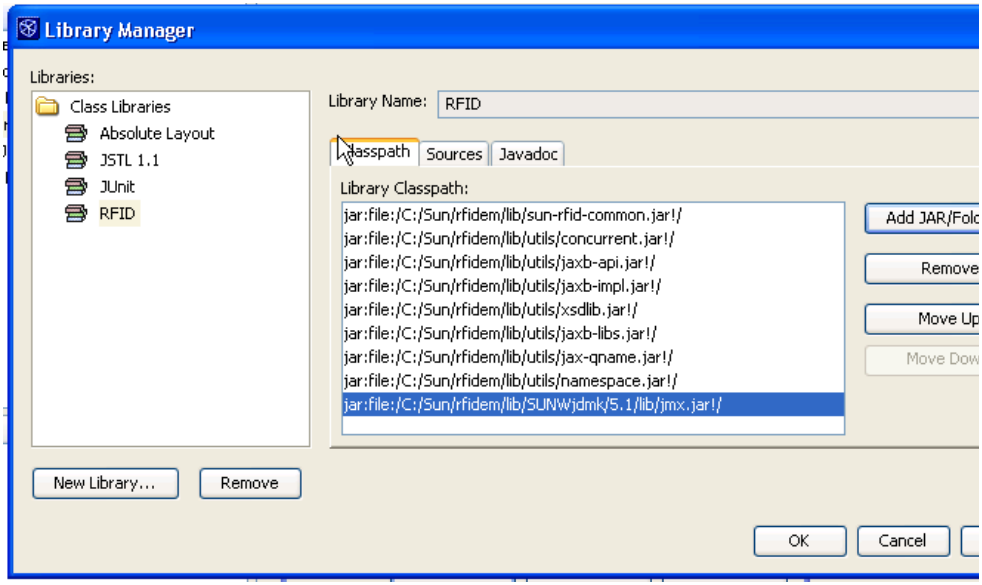
Location	Name of Required JAR file
`C:\Program Files\Sun\RFID Software\rfidem\lib`	`sun-rfid-common.jar`

`C:\Program Files\Sun\RFID Software\rfidem\lib\utils`	`concurrent.jar`
	`jaxb-api.jar`
	`jaxb-impl.jar`
	`xsdlib.jar`
	`jaxb-libs.jar`
	`jax-qname.jar`
	`namespace.jar`
	`relaxngDatatype.jar`

`C:\Sun\rfidem\lib\SUNWjdmk\5.1\lib`	`jmx.jar`

8. **When you are finished adding the JAR files to the RFID library, confirm that your Library class path is complete.**

   The dialog should appear similar to the following screen capture (from a Windows system).

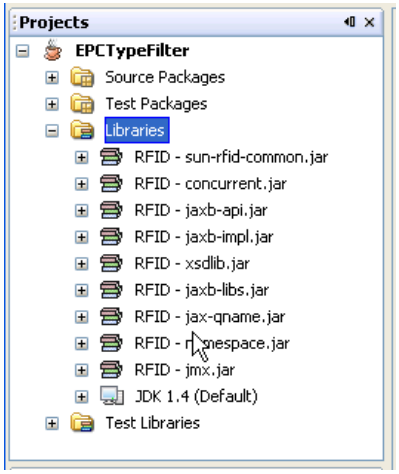


9. **Click OK.**

   This takes you back to the Add Library dialog. You only add a Library once.

10. **Select the RFID library from the list and click Add Library.**

    The libraries appear in the Projects windows as shown in the following screen capture. You should now be able to build and test successfully.



▼ To Build and Test the Sample Filter Project

1. **Right-click EPCTypeFilter project in the Projects window.**
2. **Select Clean and Build Project from the context menu.**

   The project should compile with no errors.
3. **Right-click EPCTypeFilter again and select Test Project from the context menu.**

   A successful test is indicated by the message "Build Successful" in the bottom panel.

   An unsuccessful test indicates the specific failure as the last item in the output window. If the test fails, it is likely that you do not have all of the RFID libraries added to the project. See "To Create the RFID Library for the Custom Component Examples" on page 28 for the complete list of required libraries and instructions for adding them.

   At this point, you have a sample filter that builds and test cleanly. Proceed to the next section to learn more about customizing the sample filter.

Creating a Custom Filter

This section shows how `EPCTypeFilter` was created using the NetBeans IDE. To follow these steps, you must have Netbeans 4.1 installed properly with the `com-sun-autoid-toolkit.nbm` module installed and the RFID libraries configured as described in the previous section, "Setting Up Your NetBeans Environment" on page 21.

This section contains the following procedures:

- To Customize the Sample Filter
- To Compile the Customized Filter
- To Modify and Run the JUnit Test

Understanding the Sample `EPCTypeFilter`

`EPCTypeFilter` is an example filter that screens for specific EPC types and sizes. It is a real-world example of a filter that you can usefully deploy. It contains enough detail to enable you to use the ideas on filter development of all types.

`EPCTypeFilter` filters out identifiers that do not match the set of EPC types that are configured in the filter. For example, if the `EPCTypeFilter` is configured to accept SGLN Identifiers (identifiers of class `com.sun.autoid.identity.EPC_SGLN_64` or `com.sun.autoid.identity.EPC_SGLN_96`), then all `Identifier` objects that are not of type SGLN are excluded from the filter's output.

Similarly, `EPCTypeFilter` filters out `Identifier` objects that do not match the configured EPC sizes. So if `EPCTypeFilter` is configured to accept only 96-bit tags, then all `Identifier` objects representing tags of other sizes are excluded from the filter's output.

You can use multiple rules. You can specify that the filter only accept SGLN and SSCC Identifiers from 96-bit tags. The result is that `Identifier` objects of class `com.sun.autoid.identity.EPC_SGLN_96` and `com.sun.autoid.identity.EPC_SSCC_96` pass through `EPCTypeFilter`.

The first step in creating a functional `EPCTypeFilter` example is to define the properties that are needed for configuring the filter. The filter description makes it clear that you need two properties:

- A property to specify the EPC types accepted by the filter
- A property to specify the EPC sizes accepted by the filter

It is good practice to add a description of these types to the Javadoc comments. The procedure starts by completing the class documentation.

▼ To Customize the Sample Filter

1. **Launch NetBeans.**
2. **If the `EPCTypeFilter` project is not open, open it.**

3. **Modify the class documentation to include text that describes the filter's functionality and the new configuration properties that you are adding to** `EPCTypeFilter`**.**

   a. **Expand the Source Packages node in the Projects window.**

   b. **Expand the com.mycompany node.**

   c. **Double-click** `EPCTypeFilter.java`**.**

      The source code appears in the Source Editor of the IDE.

d. Modify the class documentation to look like the following:

CODE EXAMPLE 2-1 Example Class Documentation

```
/*
 * The EPCTypeFilter filter plugs into the Sun Microsystems RFID
 * Event Manager. Filters are situated between input devices and
connectors
 * (also known as loggers) and are used to modify or remove identifiers
 * from a stream of identifiers coming from an input device (usually an
 * RFID reader).
 *
 * The EPCTypeFilter passes only specific types of EPCS, which are
 * user configurable. This filter identifies the type of identifier, and
 * if it is in the set of configured EPCTypes and has one of the configured
 * EPCSizes the Identifier passes through the filter.
 *
 * This filter assumes that all specified EPCSizes apply to all EPCTypes,
 * and does rigorous checking to ensure that there is no misconfiguration.
 * As a result, it may be necessary to configure two or more EPCTypeFilter
 * instances in combination to achieve the desired effect. This filter
 * recognizes events of type IdentifierEvent, IdentifierListEvent
 * and DeltaEvent, and outputs events of the same types.
 * If an event is passed to this filter that is not recognized by the
 * filter, the default behavior is to throw an exception.
 * This behavior can be overridden by setting the DieOnUnknownEvent
 * property to "false".
 *
 * Properties for EPCTypeFilter are as follows:
 * LogLevel - set the logging level see java.util.logging.level
 * EPCTypes - 'DoD', GIAI', 'GID', 'GRAI', 'SGLN', 'SGTIN', 'SSCC',
 * 'TYPEIII', 'TYPE1'
 *
 * EPCSizes - '96', '64' (default is '64,96')
 * DieOnUnknownEvent - default = true if this filter should throw an
 * exception if an unrecognized event is passed in. The value
 * should be false otherwise.
*/
```

As you can see, the properties, `EPCTypes` and `EPCSizes`, are used to identify the types and sizes of EPCs that pass through `EPCTypeFilter`.

Note – As you add the necessary code modifications, use the NetBeans Source Editor option Reformat Code to keep the code formatted properly. To do this, right-click in the background of the Source Editor and select Reformat Code from the context menu.

---

4. **Add a `Map` object to hold the configuration.**

   Because the filtering process uses this field, it is important that the information be stored in a manner that enables the filter to quickly determine whether or not a specific `Identifier` object should pass through or be excluded. The example uses a `Map` object for this purpose. The key is the `Identifier` object's class. A Boolean value indicates whether the class is passed through the filter. This implements the essence of the filtering mechanism as simple, fast look up. Add the `epcMap` field at the end of the list of fields near the top of the class as follows:

   - **After this code**:

TABLE 2-4 Unknown Event Exception

```
/**
* Specifies the behavior when an unknown event type is passed to this
* filter. The default behavior is to throw an exception. This can be
* overridden by setting the "DieOnUnknownEvent" property to false.
*/
private boolean dieOnUnknownEvent = true;
```

   - **Add this code**:

TABLE 2-5 Add `epcMap` Object

```
/**
* A map that contains EPCS of interest. This map associates a Class
* object with a Boolean value indicating whether we are interested in it
* or not.
*/
private Map epcMap = new HashMap();
```

5. **Now initialize the `epcMap` object.**

   This code creates entries in the `epcMap` for every available type of EPC. Initially all entries are set to `Boolean.FALSE`, indicating that none of the EPCs are accepted. In a subsequent step, the filter reads the `Properties` object passed to the constructor and directs that information to the initialization method so the appropriate entries are set to `Boolean.TRUE`, indicating that they should pass through the filter. Add the `initializeEPCMap` method after the constructor as follows:

   - **After this code**:

CODE EXAMPLE 2-2 Constructor Code.

```
/*
* Log the properties if the logger is appropriately configured
*/
if (logger.getLevel().intValue() <= Level.CONFIG.intValue()){
```

CODE EXAMPLE 2-2 Constructor Code.

```
        Enumeration e = properties.propertyNames();
         while (e.hasMoreElements()) {
                 String propertyName = (String)e.nextElement();
                String value = (String)properties.getProperty(propertyName);
                 logger.log(Level.CONFIG, "EPCTypeFilter {0}={1}",
                 new Object[] {propertyName, value});
         }
}
```

■ **Add this code**:

CODE EXAMPLE 2-3 Initialize the `epcMap` Object

```
/**
* Initializes the EPC map, which maps a Class instance to a Boolean,
* indicating our interest in the EPC type.
*
* @param epcTypes the set of types we are interested in
* @param sizes the set of sizes we are interested in
*/
private void initializeEPCMap(Collection epcTypes, Collection epcSizes)
 throws ClassNotFoundException {
 this.epcMap.put(com.sun.autoid.identity.DoD_64.class, Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.DoD_96.class, Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_GIAI_64.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_GIAI_96.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_GID_96.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_GRAI_64.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_GRAI_96.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SGLN_64.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SGLN_96.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SGTIN_64.class,
                 Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SGTIN_96.class,
                 Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SSCC_64.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_SSCC_96.class,Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_TYPEIII_64.class,
                 Boolean.FALSE);
 this.epcMap.put(com.sun.autoid.identity.EPC_TYPEI_64.class,
                 Boolean.FALSE);

/*
 * Now go through and set the EPC classes we are interested in to True.
 */
Iterator types = epcTypes.iterator();
   while (types.hasNext()) {
      String type = (String)types.next();
      Iterator sizes = epcSizes.iterator();
      while (sizes.hasNext()) {
          String size = (String)sizes.next();
          Class epcClass = null;
          try {
               epcClass =
                   Class.forName("com.sun.autoid.identity.EPC_" + type +
                                 "_" + size);
```

CODE EXAMPLE 2-3 Initialize the `epcMap` Object *(Continued)*

```
            } catch (ClassNotFoundException cnfe) {
                /*
                 * There is one EPC that doesn't fit the above
                 * template.
                 */
                epcClass =Class.forName("com.sun.autoid.identity."
                                        + type + "_" + size);
          }
           this.epcMap.put(epcClass, Boolean.TRUE);
           this.logger.log(Level.CONFIG, "EPCTypeFilter adding EPC Type
                          {0} of size {1}",
                  new Object[] {type, size});
      }
    }
}
```

The sample filter takes advantage of the format of the EPC class names to populate the map. The form holds true, except for the `com.sun.autoid.identity.DoD_64` and `com.sun.autoid.identity.DoD_96` classes. These are dealt with by catching the `ClassNotFound` exception that results from trying to instantiate an `EPC_DoD_64` or `EPC_DoD_96` class, and trying the form that is not preceded by "EPC_".

6. **To complete the initialization of the map, read the `EPCTypes` and `EPCSizes` properties from the `Properties` object that is passed to the constructor.**

   The types and sizes are represented as comma-separated `Identifier` objects matching the name and size of the allowed identifiers. For example, if the `EPCTypes` is set to "`SGLN,DoD`" and `EPCSizes` is set to "`64,96`", `EPCTypeFilter` is configured to accept all `Identifier` objects of class as follows:

   - `com.sun.autoid.identity.EPC_SGLN_64`
   - `com.sun.autoid.identity.EPC_SGLN_96`
   - `com.sun.autoid.identity.DoD_64`
   - `com.sun.autoid.identity.DoD_96`

   Knowing that the property is a `String` containing a comma-separated list and the `initializeEPCMap` method accepts a collection, the final initialization code converts the comma-separated list into a collection and call the initialization method. Add this code to the constructor:

   - **After this code**:

```
/**
* Instantiate a EPCTypeFilter filter with the specified properties.
*
* @param properties the Properties object to use for initialization of the
* filter
*/
public EPCTypeFilter(Properties properties) {
        super(EPCTypeFilter.MY_NAME, properties);
      /*
       * Set the appropriate log level based on the "LogLevel" property.
       */
       String logLevel = properties.getProperty("LogLevel");
       if (null != logLevel) {
             this.setLogLevel(logLevel);
       }
       String dieOnUnknown = properties.getProperty("DieOnUnknownEvent");
       if (null != dieOnUnknown) {
           this.dieOnUnknownEvent =
                    Boolean.valueOf(dieOnUnknown).booleanValue();
      }
      /*
      * Log the properties if the logger is appropriately configured.
      */
     if (logger.getLevel().intValue() <= Level.CONFIG.intValue()) {
           Enumeration e = properties.propertyNames();
           while (e.hasMoreElements()) {
              String propertyName = (String)e.nextElement();
              String value = (String)properties.getProperty(propertyName);
              logger.log(Level.CONFIG, "EPCTypeFilter {0}={1}",
              new Object[] {propertyName, value});
                                              }
     }
}
```

- **Add this code**:

CODE EXAMPLE 2-4 Code to Read the EPCTypes and EPCSizes Properties

```
/**
 * Read the interesting identifier types and sizes and prepare the map
 * accordingly.
 */
 String epcTypes = properties.getProperty("EPCTypes");
 if (null == epcTypes) {
     throw new IllegalArgumentException("\"EPCTypes\" property must be
                                        configured");
```

CODE EXAMPLE 2-4 Code to Read the EPCTypes and EPCSizes Properties *(Continued)*

```
}
Collection epcNames = new LinkedList();
StringTokenizer tokenizer = new StringTokenizer(epcTypes, " ,\t\n\r\f");
while (tokenizer.hasMoreTokens()) {
    String epcType = tokenizer.nextToken();
    epcNames.add(epcType);
}
/*
 * Read the size(s) that we are interested in.
 */
String sizes = properties.getProperty("EPCSizes");
if (sizes == null) {
    /*
     * By default, look for 64 and 96 bit tags of this type.
     */
    sizes = "64,96";
}
Collection epcSizes = new LinkedList();
tokenizer = new StringTokenizer(sizes, " ,\t\n\r\f");
while (tokenizer.hasMoreElements()) {
    String size = tokenizer.nextToken();
    epcSizes.add(size);
}
try {
    this.initializeEPCMap(epcNames, epcSizes);
} catch (ClassNotFoundException cnfe) {
    throw new IllegalArgumentException("Could not initialize the
                                        EPCTypeFilter: "
          + cnfe.getMessage());
 }
}
```

7. **Add code to modify the** `processOneIdentifier` **method to filter out all** `Identifier` **objects that were not specified in the configuration.**

   This modification requires getting the class of the `Identifier` object and checking the `epcMap` contents to see whether the `Identifier` object should pass through or be rejected.

   Change the `processOneIdentifier` method as follows:

   - **After this code**:

```
* TODO: Modify this method to filter events.
*
* @param identifier the Identifier to examine for filtering
* @param action one of TAG_SEEN, TAG_IN, TAG_OUT, specifying the tag
* activity @param identifiersToSend a Collection of identifiers that will
* be sent
*/
private void processOneIdentifier(Identifier identifier, int action,
         Collection identifiersToSend) throws Exception {
    /*
     * To filter out this identifier, simply don't add it to the list. You
     * can also add properties to the identifier with code similar to:
     * identifier.addProperty("key", "value");
     */
```

- **Add this code**:

CODE EXAMPLE 2-5 Modify the Filter

```
Class identifierClass = identifier.getClass();
Boolean interesting = (Boolean)this.epcMap.get(identifierClass);
if (null != interesting && interesting.booleanValue()) {
    identifiersToSend.add(identifier);
}
```

- **And delete this code**:

```
identifiersToSend.add(identifier);
```

An `Identifier` object is accepted by adding the object to the `identifiersToSend Collection` object. Only `Identifier` objects that are added to this `Collection` object are sent. The code that sends the event can be found in the process method.

With this code modification, the `EPCTypeFilter` example is complete. The example shows the initialization of the filter, the scheme used to determine if an EPC type should be accepted or rejected by the filter, and the mechanics of filtering the `Identifier` objects.

▼ To Compile the Customized Filter

1. **Save the** `EPCTypeFilter.java` **file.**

2. **Right-click the EPCTypeFilter node in the Projects window.**
3. **Select Clean and Build Project from the context menu.**

   You see BUILD SUCCESSFUL at the bottom of the output window.

Using the Filter Template JUnit Test

A JUnit test is created as part of the filter template. This test needs to be modified because your new filter requires initialization of the `EPCTypes` and `EPCSizes` properties.

▼ To Modify and Run the JUnit Test

1. **Expand the Test Packages node in the Projects window.**
2. **Expand the com.sun.rfid.filter node.**
3. **Double-click the `EPCTypeFilterTest.java` file.**

   The JUnit test source code appears in the main window.
4. **Modify the `setUp` method to initialize the `EPCTypes` and `EPCSizes` properties.**
   - **After this code**:

```
**
* Sets up the test fixture. Called before every test case method.
*/
protected void setUp() {
        this.props = new Properties();
        this.props.put("LogLevel", LOGLEVEL_TEST);
```

   - **Add this code**:

CODE EXAMPLE 2-6 Modify the JUnit Test

```
this.props.put("EPCTypes", "SGLN,SSCC");
this.props.put("EPCSizes", "96,64");
```

You have initialized `EPCTypeFilter` to accept only `Identifier` objects of type `SGLN` and `SSCC`, of both 96-bit and 64-bit sizes. This filter now accepts `Identifier` objects of the following classes:

- `com.sun.autoid.identity.EPC_SGLN_96`

- com.sun.autoid.identity.EPC_SGLN_64
- com.sun.autoid.identity.EPC_SSCC_96
- com.sun.autoid.identity.EPC_SSCC_64

You can run the test now to see if it is working.

5. **Right-click the EPCTypeFilter node in the Projects window.**
6. **Select Test Project from the context menu.**
7. **Check the output window for the result.**

   You find that the test indicates a failure as follows:

```
Testcase:
testProcessIdentifierListEvent(com.sun.rfid.filter.EPCTypeFilter
Test): FAILED
Expected 13 identifiers expected:<13> but was:<4>
```

   This message indicates the filter is actually working. Of the 13 `Identifier` objects passed in to this filter, only four were accepted. To fix this last problem, proceed to the next step.

8. **Modify the** `testProcessIdentifierEvent` **method in the** `EPCTypeFilterTest.java` **file.**

■ **After this code**:

```
/**
* Test the processing of events. This method should be modified to send
* interesting events to the filter and ensure that the appropriate events
* get filtered or modified properly.
*/
public void testProcessIdentifierEvent() {
     this.filter.reset();
     int tagCount = this.tags.length;
     long timestamp = System.currentTimeMillis();
     try {
          for (int index = 0; index < tagCount; index++) {
               Identifier identifier = IdentifierFactory.createEPC(new
URI(this.tags[index]));
               IdentifierEvent event = new IdentifierEvent(identifier,
                       READER, timestamp, "IdentifierEvent from " +
READER);
               this.filter.postEvent(event);
          }
     } catch (Exception e) {
         fail("Unexpected failure of EPC creation");
     }
     /**
      * Wait for the identifiers to be processed.
      */
     this.pause(1000);
```

■ **Replace this code:**

```
assertEquals("Expected " + tagCount + " identifiers", tagCount,
             this.filter.getPassCount());
```

■ **With this code**:

TABLE 2-6 Modify JUnit test

```
assertEquals("Expected 4 identifiers", 4,
             this.filter.getPassCount());
```

With this modification, the unit test now specifies how many `Identifier` objects to expect in the result.

9. **Modify the** `testProcessIdentifierListEvent` **method in the** `EPCTypeFilterTest.java` **file.**

   This change again indicates how many Identifiers are expected to pass through the filter.

   Change the JUnit code as follows:

   - **After this code**:

```
/**
* Test to ensure that the EPCTypeFilter filter can handle events of type
* IdentifierListEvent.
*/
public void testProcessIdentifierListEvent() {
    this.filter.reset();
    int tagCount = this.tags.length;
    long timestamp = System.currentTimeMillis();
    try {
         IdentifierListEvent event = new IdentifierListEvent(READER,
                  timestamp, "IdentifierListEvent from " + READER);
         for (int index = 0; index < tagCount; index++) {
              Identifier identifier = IdentifierFactory.createEPC(new
                 URI(this.tags[index]));
              event.addIdentifier(identifier);
         }
          this.filter.postEvent(event);
     } catch (Exception e) {
         fail("Unexpected failure of EPC creation");
     }
     /**
     * Wait for the identifiers to be processed.
     */
    this.pause(1000);
```

   - **Replace this code:**

```
assertEquals("Expected " + tagCount + " identifiers", tagCount,
              this.filter.getPassCount());
```

   - **With this code**:

CODE EXAMPLE 2-7 Modify JUnit Test

```
assertEquals("Expected 4 identifiers", 4,
              this.filter.getPassCount());
```

10. **Modify the** `testProcessDeltaEvent` **method in the** `EPCTypeFilterTest.java` **file.**

    This modification is specific to the `DeltaEvents` method and is the same modification as in Step 8 and Step 9.

    - **After this code**:

CODE EXAMPLE 2-8 Existing JUnit `ProcessDeltaEvent` Method

```
/**
* Test to ensure that the EPCTypeFilter filter can handle events
* of type DeltaEvent.
*/
public void testProcessDeltaEvent() {
    this.filter.reset();
    int tagCount = this.tags.length;
    long timestamp = System.currentTimeMillis();
    try {
        DeltaEvent event = new DeltaEvent(READER, timestamp,
                    "DeltaEvent from " + READER);
        Collection tagsIn = new LinkedList();
        Collection tagsOut = new LinkedList();
        for (int index = 0; index < tagCount; index++) {
             Identifier identifier =
             IdentifierFactory.createEPC(new URI(this.tags[index]));
         /*
          * Even index identifiers into tagsIn; odd into tagsOut.
          */
          if (index % 2 == 0) {
               tagsIn.add(identifier);
        } else {
               tagsOut.add(identifier);
         }
       }
       event.setTagsIn(tagsIn);
       event.setTagsOut(tagsOut);
       this.filter.postEvent(event);
   } catch (Exception e) {
       fail("Unexpected failure of EPC creation");
   }
   /**
    * Wait for the identifiers to be processed.
    */
   this.pause(1000);
```

- **Replace this code:**

```
assertEquals("Expected " + tagCount + " identifiers", tagCount,
             this.filter.getPassCount());
```

- **With this code**:

CODE EXAMPLE 2-9 New JUnit `ProcessDeltaEvent` Method

```
assertEquals("Expected 4 identifiers", 4,
             this.filter.getPassCount());
```

11. **Save the `EPCTypeFilterTest.java` file and test the filter again. (See Step 5 through Step 7).**

    Look for "BUILD SUCCESSFUL" in the last line of the output window.

Integrating Custom Components With the RFID Event Manager

This section describes how to integrate the EPCTypeFilter custom filter into the RFID Event Manager. The RFID Configuration Manager enables you to connect various RFID Event Manager components (adapter, filters and connectors) to create a processing chain known as Business Processing Semantics (BPS). See *Sun Java System RFID Software 3.0 Administration Guide* for more details. For this example, you use the RFID Configuration Manager to add the EPCTypeFilter custom filter to an existing BPS, the default Demo Configuration Object.

▼ To Add the `EPCTypeFilter` Custom Filter to the `Demo` Configuration Object

1. **If you have not already done so, start the RFID Configuration Manager.**
2. **Add the new EPCTypeFilter to the RFID Configuration Manager and configure its properties by using these steps:**

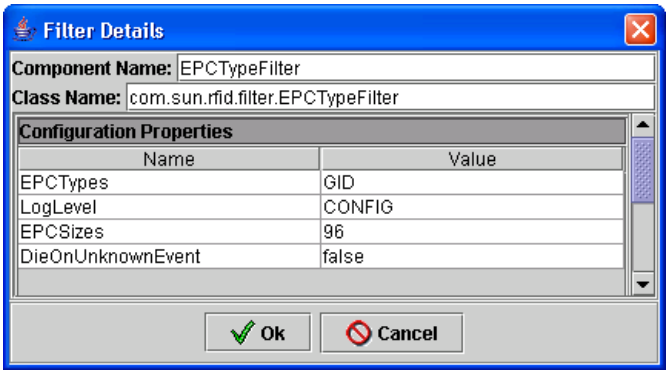
   a. **From the RFID Configuration Manager menu, choose Roles → Edit.**

      The RFID Role and Component Editor window appears.

b. **In the navigation tree, under the Roles node, select the `Demo` role.**

c. **Choose Filter → New.**

Set up the filter properties as shown in the following screen capture.



d. **Click Ok.**

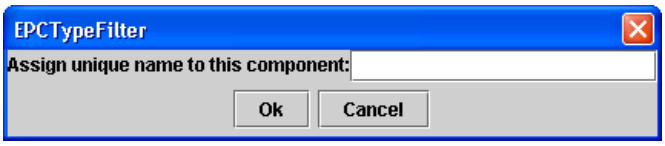
The EPCTypeFilter filter is added to the navigation tree in the left pane.

3. **Add the EPCTypeFilter to the Demo Role by following these steps:**

a. **In the navigation tree, expand the Components node and expand the Filters node.**

b. **Right-click EPCTypeFilter and choose Add to Role from the contextual menu.**

A dialog box appears and prompts you to assign a unique name to this component.

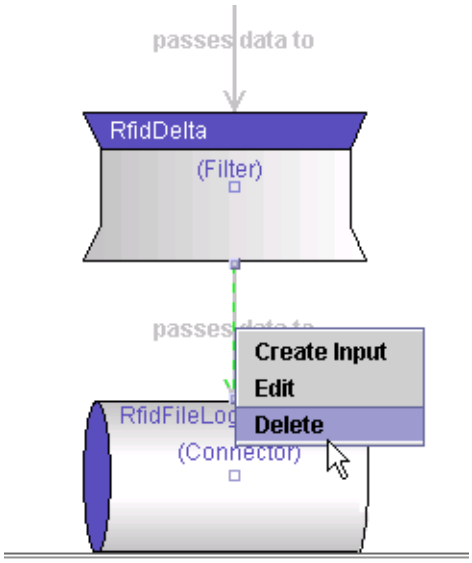


c. **Type RfidEPCTypeFilter and click Ok.**

The RfidEPCTypeFilter component appears in the drawing pane.

4. **Connect the components in the necessary order by following these steps:**

a. **Select and right-click the arrow that connects the Rfid Delta filter and the RfidFile logger components.**



b. Choose Delete from the contextual menu.

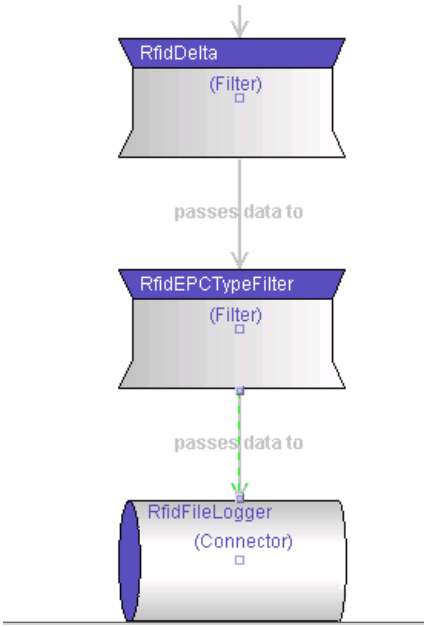
The arrow disappears.

c. Connect the RfidDelta component to the RfidEPCTypeFilter component.

To do so, click the port (the small square at the center of each component) on the RfidDelta component and drag a line to the RfidEPCTypeFilter component.

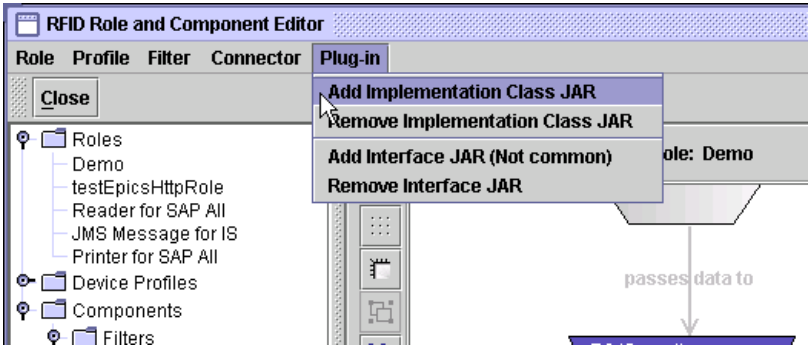
d. Connect the RfidEPCTypeFilter to the RfidFile logger component.

Click the port on the RfidEPCTypeFilter component and drag a line to the RfidFile logger component. The drawing pane shows the new component flow as seen in the following screen capture.



5. **Add the JAR file for the new EPCTypeFilter component by following these steps:**

   a. **From the RFID Role and Component Editor menu, choose Plug-in → Add Implementation Class JAR.**



   b. **Use the file chooser to navigate to the EPCTypeFilter.jar, select the JAR file and click Ok.**

6. **Close the RFID Role and Component Editor window.**

7. **Delete any existing Demo Configuration Object.**

   This is necessary so that you can create a new Demo Configuration Object to pick up the changes that you made to the Demo role.

8. **Create a new `Demo` Configuration Object by using the following steps:**

a. **Choose Configuration → New.**

b. **Select `Demo` as the Base Role.**

c. **In the Configuration Object Name field, type EPCTypeFilter.**

d. **Select the PMLReader as the Input Point and click Ok.**

9. **Choose File → Save and save your new EPCTypeFilter Configuration Object.**

   Now you are ready to start the RFID Event Manager and use the new Demo Configuration Object.

Creating a Custom Connector

The Sun Java System RFID Software Toolkit NetBeans plug-in (.nbm), `com-sun-autoid-toolkit.nbm`, also contains a template called `RfidConnector`. You can use it to simplify the creation of custom connectors for the RFID Software.

To install the RFID Software Toolkit NetBeans plug-in, you first download and install the NetBeans 4.1 IDE. Then, if you have not already done so, you need to set up your development environment using the following procedures:

- "To Download and Install NetBeans" on page 22
- "To Download and Install the RFID Software Toolkit" on page 22

▼ To Create a Sample Connector Project

1. **Follow the steps in the procedure "To Set Up the Example Filter Project" on page 25 with one change. In Step 3, select RfidConnector instead of RfidFilter.**

   For the purposes of this example, name your sample connector project, `RFIDConnector`.

2. **Confirm the default name for the connector, `MyConnector`.**

3. **If you have not already done so, perform the procedure "To Create the RFID Library for the Custom Component Examples" on page 28 to create the necessary RFID library.**

   This library is used for both the sample filter and the sample connector.

4. **Now, make the following changes to the sample code so that the project will build correctly:**

   a. **Open the file, `MyConnector.java`.**

b. In the IDE's Source editor, search for the following line of code and remove the line of code from the java source file:

```
tagsOut = new LinkedList();
```

c. Search for the following line of code:

```
this.passCounter += identifierCount;
```

d. Change this line to the following:

```
this.identifierCounter += identifierCount;
```

e. Search for the following line of code:

```
private long identifierCounter = 0L;
```

f. Just after this line of code, add the following code:

```
private long discardCounter = 0L;
```

5. **Right-click the RFIDConnector project in the Projects window.**
6. **Select Clean and Build from the context menu.**

   The project should compile with no errors.
7. **Now you can read through the comments in this code and modify it to create your custom connector.**

CHAPTER 3

Using RFID Device Client APIs

This chapter describes how to use the Java APIs for reader and printer clients.

Each Execution Agent provides one or more reader services. A reader service is a specialized web service that communicates with the RFID device, processes the information as directed by the RFID Event Manager configuration object, and communicates with the RFID Information Server or third-party Enterprise Resource Planning (ERP) systems. Configuration objects are defined for each specific reader using the RFID Configuration Manager. See the *Sun Java System RFID Software 3.0 Administration Guide* if you are not familiar with the concept of a configuration object and how they are defined.

In the RFID Event Manager, the reader service performs the work of gathering RFID tag events. The reader service communicates the state of its components to the management service (another specialized web service). The communication from the reader service to the management service enables the RFID Management Console to monitor the components.

Previous releases of the RFID Software required developers to be familiar with Jini programming in order to discover the reader service in the Jini lookup server and to invoke the service's device access methods.

A set of reader and printer client APIs has been implemented to hide some of the complexity of working directly with Jini services.

This chapter includes the following topics:

Implementation of the `ReaderClient` API

The implementation of the `ReaderClient` API is contained in the `com.sun.autoid.util.ReaderClient` class and is packaged in the `sun-rfid-common.jar` JAR file.

This section covers the following topics:

Reader Client Constructor Parameters

There are many ways of instantiating a `ReaderClient` constructor. The various parameters for the `ReaderClient` constructors are described in the following sections:

Reader Client Groups Parameter

During installation of the RFID Event Manager Control Station, you are prompted to enter a group name. All readers configured for this Control Station are associated by that group name. If a second Control Station is started, a unique group name for this Control Station must be assigned. All readers in this Control Station are then associated by this new group name. The `ReaderClient` class uses this group name to help narrow the search for a given reader. If no group name is specified, then all groups are searched.

Many of the `ReaderClient` constructors take a `String[] groups` argument. This argument is a `String` array of group names. If all groups are to be searched, then the argument is null. You can use the `ReaderClient method String[] getGroups(String groupStr)` to create this array of group names. The

`getGroups ()` method takes a `String` of group names and returns a `String` array of group names. Each group name is separated from the next group name by a space, a tab or a comma, with a special case for *all* or *none*.

Reader Client `Locators` Parameter

By default, a reader client can only find those readers that are running on RFID Event Manager Execution Agents and Control Stations within the same subnet. You can extend the search for readers outside this subnet by adding locators. A locator is defined in terms of a URL format. For example, `jini://`*hostname*`:`*port.*

The variable, *hostname*, is the host name or IP address of a machine that is running the RFID Event Manager Control Station that manages the reader services that you want to add to the reader search list.

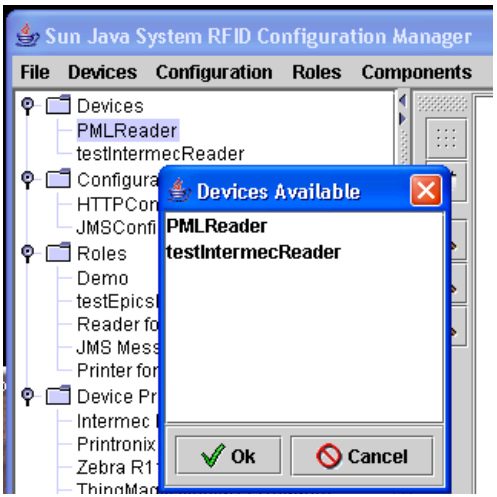
The, variable, *port*, is the Jini lookup server port number. This port number is optional when defining a new locator, unless you customized the port number during the installation of the RFID Event Manager. If you add a Jini location and the port number is not specified, the default Jini lookup server port, 4160, is used. If you customized the Jini lookup server port during installation, you must specify the same port number when you add a locator to this Jini lookup server. The port number must match the one used during installation.

Many of the `ReaderClient` constructors take a `LookupLocator[] locators` argument. This should be a `LookupLocator` array of locator objects or null if only the local subnet is to be searched. You might also use the `ReaderClient` method `LookupLocator[] getLocators(String locatorStr)` to create this array of locator objects. The `getLocators ()` method takes a `String` of locator values in the URL format Each locator is separated from the next location by a space, a tab or a comma, and returns an array of `LookupLocator` objects.

Reader Client `readerName` Parameter

All `ReaderClient` constructors require you to supply a reader name as an input. This reader name is the name of the specific device as configured in the RFID Configuration Manager.

For example, you might specify PMLReader or testIntermecReader as the `readerName` parameter to the `ReaderClient` constructor to reach one of the two devices shown in the following screen capture. To create your own specific readers, consult the *Sun Java System RFID Software 3.0 Administration Guide*. The `testIntermecReader` shown in the screen shot is for demonstration purposes and does not represent a real reader.



Reader Client `eventID` Parameter

An `eventID` is a unique number used by the RFID Software to identify *remote event producers* (`REProducer`) for tag events. Client applications use this identifier to connect to a specific `REProducer` component. In RFID Software 3.0, each reader automatically creates a `REProducer` component using a randomly generated unique number.

Note – In RFID Software 2.0, you had to use a separate `REProducer` component in the reader's processing chain specifically for this purpose and then use the `eventID` to locate the `REProducer`. It is no longer necessary to include a `REProducer` connector for this purpose.

Generally, you should locate the reader's internal `REProducer` by using the reader's physical name. If you want to control the assignment of the `eventID` number, you define a `Handle` property in the reader's configuration.

Some of the `ReaderClient` constructors take an `eventID` parameter. If an `eventID` is specified, then the reader client connects to the default `REProducer` and matches a reader with this `eventID`.

The `eventID` may be null. If the `eventID` is null, then the reader's event producer is found by using the reader's physical name. The reader physical name is the name assigned to the reader when you define the physical device using the RFID Configuration Manager. See "To Define the RFID System Physical Devices" in the *Sun Java System RFID Software 3.0 Administration Guide*.

When an `eventID` is specified, the reader adapter configuration must contain a `Handle` property. To establish a connection to the reader's event producer, the value of the `Handle` property must match the reader client `eventID` parameter.

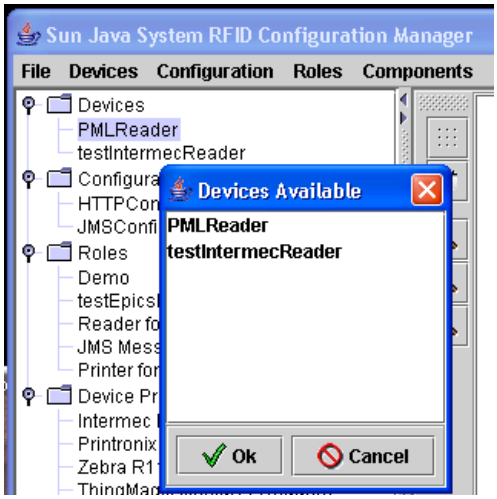
You use the RFID Configuration Manager to add a `Handle` property to a reader adapter. The reader adapter configuration is referred to as a *device* in the RFID Configuration Manager. `Handle` is not a default property for a device, so you must add the `Handle` property using the RFID Configuration Manager. See "To Add a `Handle` Configuration Property to a Device" on page 57.

▼ To Add a `Handle` Configuration Property to a Device

Prerequisite – This procedure assumes that you are familiar with how to use the RFID Configuration Manager. See the *Sun Java System RFID Software 3.0 Administration Guide* for more information on how to define and configure devices in the RFID network.

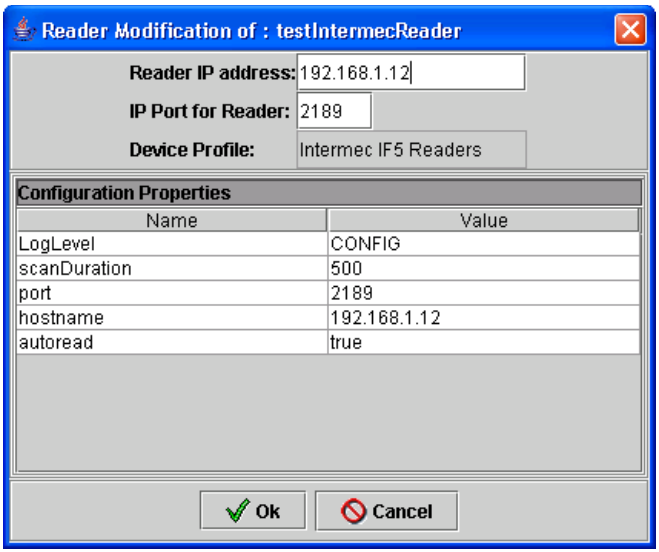
1. **If you have not already done so, start the RFID Configuration Manager.**
2. **From the RFID Configuration Manager menu, choose Devices → Edit.**

   A dialog listing the available devices appears.



3. **Select the specific reader and click Ok.**

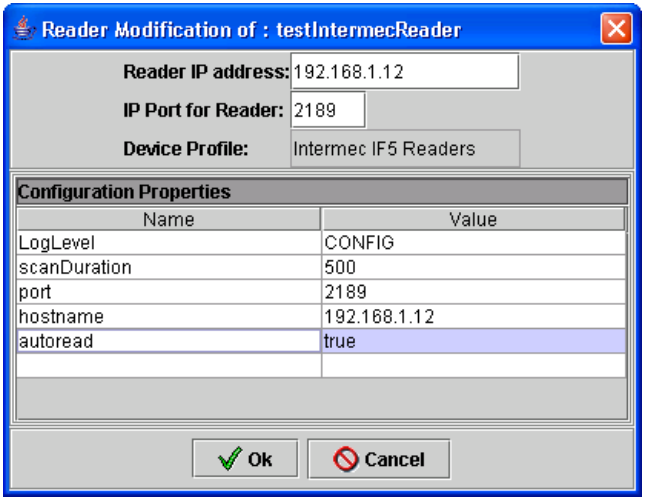
   For the purposes of this example, the `testIntermecReader` was selected. An edit dialog box for the device appears as shown in the following screen capture.



4. **Select any configuration property name field.**

5. **Right-click and choose Add Property from the context menu.**

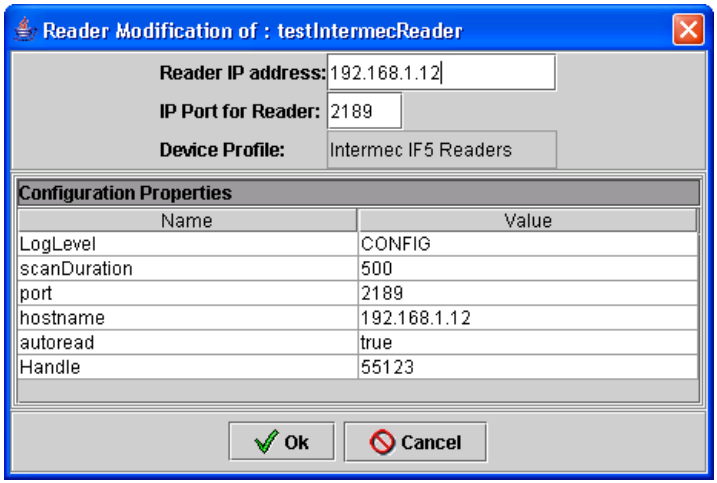
   A blank row appears enabling you to add the Name and Value for the new property.



6. **In the new configuration property Name field, type Handle.**

7. **In the Value field, type 55123 and click Ok.**

   The new configuration property is added to the device. The following screen capture shows the newly added configuration property of `Handle` with a value of 55123 added to the `testIntermecReader`.



8. **Choose File → Save to save your change.**

Using the configuration that is shown in this example procedure, use the `Handle` value of `55123` to create a `ReaderClient` as shown in the following code example.

CODE EXAMPLE 3-1 Creating a `ReaderClient`

```
public class MyReaderClient implements EMSEventListener {
  public static void mainString[] args) {
    System.setSecurityManager(new RMISecurityManager());
    long eventID = 55123;
    ReaderClient = new ReaderClient("testIntermecReader", eventId, this);
  }
    ...
}
```

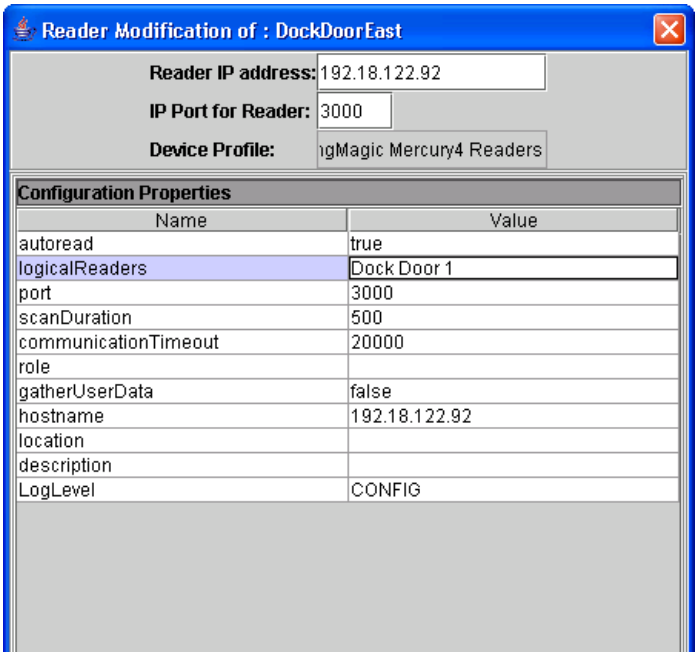
Reader Client `logical` Parameter

One of the `ReaderClient` constructors enables you to specify the `String` logical. This `String` refers to a group of logical readers that have been defined during the RFID Event Manager configuration. For example, a single dock door may have several readers positioned strategically around the door to enable the best read possible as pallets pass through the door. It might be useful to group these readers into a logical group that can be tracked as a single unit. The `logical` parameter enables you to discover a reader when the reader is part of a specific logical group.

To use the `logical` parameter, you must define a logical group using the RFID Configuration Manager. For example, you might create a logical group named, Dock Door 1 that contains two readers, DockDoorEast and DockDoorWest. This enables

you to refer to both of these readers using the logical name of Dock Door 1. Use the RFID Configuration Manager to configure two readers with a `logicalReaders` property of Dock Door 1.

A ThingMagic Mercury4 reader named DockDoorEast that is configured with a `logicalReaders` property, Dock Door 1, appears in the following screen capture from the RFID Configuration Manager. You would configure the reader at DockDoorWest in the same way.



Note – When you configure a device to be part of a `logicalReaders` group, if the `logicalReaders` property does not already appear in the Configuration Properties list, you can add it to the device configuration. Use a procedure similar to that described for adding the `Handle` property. See "To Add a `Handle` Configuration Property to a Device" on page 57.

EMSEventListener

The `ReaderClient` API also enables you to specify an `EMSListener` to receive reader events. If specified, this listener is registered with the default reader event producer and receives events using the `postEvent()` method. The `EMSEventListener` can be null.

ReaderClient API Reference

All of the public ReaderClient APIs are listed in the following table.

TABLE 3-1 ReaderClient Interfaces

Constructors and Methods	Description
`ReaderClient(String readerName)`	• Creates a new `ReaderClient` instance matching on the `readerName` in ALL groups. • The default event producer is used. • No remote listener is registered. If one is used, it must be manually registered using the `addEMSEventListener()` method.
`ReaderClient(String readerName, String[]groups)`	• Creates a new `ReaderClient` instance matching on the reader name in `groups` specified by the `groups` parameter. See `getGroups ()` method for details. • The default Event Producer is used. • No remote listener is registered. If one is used, it must be manually registered using the `addEMSEventListener()` method.
`ReaderClient(String readerName, String[]groups, Locator[] locators)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the `groups` parameter. See `getGroups ()` method for details. • The local subnet is searched in addition to hosts specified by the `locators` parameter. See `getLocators()` method for details. • The default Event Producer is used. • No Remote Listener is registered. If one is used, it must be manually registered using the `addEMSEventListener()` method.
`ReaderClient(String readerName, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in ALL groups. • The default Event Producer is used • The specified listener is registered to the default reader event producer.

TABLE 3-1 `ReaderClient` Interfaces *(Continued)*

Constructors and Methods	Description
`ReaderClient(String readerName, long eventID, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in ALL groups. • The specified listener is registered to the default reader event producer for a reader adapter that has explicitly set its `Handle` id and must match the `eventID` parameter.
`ReaderClient(String readerName, long eventID, String[] groups, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the groups parameter. See `getGroups ()` method for details. • The default Event Producer is used. • The specified listener is registered to the default reader event producer for a reader adapter that has explicitly set its `Handle` id and must match the `eventID` parameter.
`ReaderClient(String readerName, String[]groups, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the groups parameter. See `getGroups ()` method for details. • The default Event Producer is used. • The specified listener is registered to the default reader event produce to receive reader events.
`ReaderClient(String readerName, String[] groups, Locator[] locators, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the groups parameter (see `getGroups ()` method for details. • The local subnet is searched as well as any additional hosts as specified by the `locators` parameter. See `getLocators()` method for details. • The default Event Producer is used. • The specified listener is registered to the default reader event producer to receive reader events.

TABLE 3-1 `ReaderClient` Interfaces *(Continued)*

Constructors and Methods	Description
`ReaderClient(String readerName, long eventID, String[] groups, Locator[] locators, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the groups parameter (see `getGroups ()` method for details. • The local subnet is searched in addition to hosts specified by the `locators` parameter. See `getLocators()` method for details. • The specified listener is registered to the default reader event producer for a reader adapter that has explicitly set its `Handle` id and must match the `eventID` parameter.
`ReaderClient(String readerName, long eventID, String logical, String[] groups, Locator[] locators, EMSEventListener listener)`	• Creates a new `ReaderClient` instance matching on the Reader Name in groups specified by the `groups` parameter. See `getGroups ()` method for details. • The local subnet is searched in addition to hosts specified by the `locators` parameter. See `getLocators()` method for details. • The specified listener is registered to the default reader event producer for a reader adapter that has explicitly set its `Handle` id and must match the `eventID` parameter. • See "Reader Client `logical` Parameter" on page 59 for details of the `logical` parameter.
`terminate`	Terminates discovery of the reader.
`checkReaderStatus`	Returns true if the reader has been discovered and is connected to its device, else returns false.
`getLastList`	Gets the last tag list, the tag list is not cleared. A subsequent call to `getLastList` returns the same list, unless the Remote Event Producer has caused the list to be updated.
`takeLastList`	Takes the last tag list and clears it afterwards. A subsequent call to `getLastList` or `takeLastList` does not return the same list. The subsequent call returns a new list if it has been updated by the Remote Event Producer, or return null, if no update has taken place.

TABLE 3-1 `ReaderClient` Interfaces *(Continued)*

Constructors and Methods	Description
`takeLastList (msecs)`	Takes the last tag list and clears it afterwards. A subsequent call to `getLastList` or `takeLastList` does not return the same list. The subsequent call returns a new list if the last has been updated by the Remote Event Producer, or returns null, if no update has taken place. This method waits for a period of time for the list to be updated if the list is empty when the call is first made.
`getReader`	Get the reader interface for this reader. Might return null if the reader has not yet been discovered.
`getReader (msecs)`	Get the reader interface for this reader. Might return null if the reader has not yet been discovered by the specified wait time.
`addEMSEventListener`	Add an `EMSEventListener` to the Remote Event Producer to be notified of reader events.
`removeEMSEventListener`	Remove an `EMSEventListener` from the Remote Event Producer canceling notification of reader events

Building a Sample Reader Client Program

A sample reader client program is included in the Sun Java System RFID Software Toolkit. After unzipping the `RfidToolkit.zip` file into a directory of your choice, *toolkit-dir*, you can find `SampleTagReaderClient.java` in the following directory: *toolkit-dir*`/samples/readerAccess/standAlone`.

Before running `SampleTagReaderClient`, confirm the following prerequisites:

- The RFID Event Manager has been started.
- You can see that the Tag Viewer is receiving tag events from the RFID Event Manager.

▼ To Set Up the Sample Reader Client Environment

1. **Change to the directory containing the reader client sample program.**

   For example, on a Solaris system where *toolkit-dir* represents the directory where you downloaded and unzipped the RFID Software Toolkit zip file.

   ```
   cd toolkit-dir/samples/readerAccess/standAlone
   ```

2. **Verify that the `build.properties` file is correct for your installation.**

   You might need to modify the `rfid.home` property to point to your specific RFID Event Manager installation.

3. **Verify that the `build.xml` file target `sampletagreader` contains the correct `jvmarg` parameter values for your environment as follows.**

   - `-Dcom.sun.autoid.ReaderName` – The name of the reader you want to access.
   - `-Dcom.sun.autoid.JiniGroup` – The name of the group that you specified during installation of your RFID Event Manager. The default value is AutoID-*hostname*.
   - `-Dcom.sun.autoid.TagBitSize` – The size of the tag that you want to write.

▼ To Run the Sample Reader Client Program

1. **Change to the directory containing the reader client sample program.**

   For example, on a Solaris system where *toolkit-dir* represents the directory where you downloaded and unzipped the RFID Software Toolkit zip file.

   ```
   cd toolkit-dir/samples/readerAccess/standAlone
   ```

2. **Run the reader client using the `ant sampletagreader` target.**

   ```
   > ant sampletagreader
   ```

3. **Run the writer client using the `ant sampletagwriter` target.**

   ```
   > ant sampletagwriter
   ```

Explaining the Sample Reader Client

To create a reader client, you first must set the system security manager and a security policy. This is necessary because you use Jini network technology to discover the reader using the `ReaderClient` APIs. This step is shown in the following code example.

CODE EXAMPLE 3-2 Setting an RMI Security Manager and the Security Policy

```
// You must first create an RMI Security manager
System.setSecurityManager(new RMISecurityManager());
// Must also set the security policy
 if(System.getProperty("java.security.policy") == null)
 System.setProperty("java.security.policy", "./policy.all");
```

Set the `codebase` property so that the `ReaderClient` can register a notification call back with the Jini lookup service as follows:

CODE EXAMPLE 3-3 Setting the `codebase` Property for the Jini Lookup Service

```
// Must set code base so that the ReaderClient can register a
// notification call back with the Jini Lookup Service
if(System.getProperty("java.rmi.server.codebase") == null)
    System.setProperty("java.rmi.server.codebase",
    "http://localhost:52493/sdm-dl.jar");
```

Then use one of the many ReaderClient constructors to find the reader.

When you install the RFID Event Manager, you supply a Jini group name. By default, this name is `AutoID-`*hostname*. The default RFID Event Manager installation also configures the PMLReader adapter. The PML simulator software simulates tags being sent to the PMLReader adapter. To start the PML simulator software, run the `pmlreader` script.The script can be found in the *rfid-install-dir*`/bin` directory. See the *Sun Java System RFID Software 3.0 Administration Guide* for more details. Using the default installation configuration and a host name of `myHost`, the following code example shows how to create a `ReaderClient` that discovers the PMLReader.

CODE EXAMPLE 3-4 Finding the PMLReader Reader in the `AutoID-myHost` Group

```
String readerName =  "PMLReader";
String groupStr = "AutoID-myHost";
String locatorStr = null;
ReaderClient readerClient = null;
```

CODE EXAMPLE 3-4 Finding the PMLReader Reader in the `AutoID-myHost` Group *(Continued)*

```
try {
          readerClient = new ReaderClient(readerName,
              ReaderClient.getGroups(groupStr),
              ReaderClient.getLocators(locatorStr));
         // Now get the actual reader implementation
         Reader reader = client.getReader(3*1000);
    }
}
```

After creating the `ReaderClient` object, you might use it to get the actual reader interface. The reader interface can be used to access and control the reader.

For example, the following code samples illustrate how to get a list of tags from the reader and how to check the status of a reader.

CODE EXAMPLE 3-5 Printing a List of Tags From the PMLReader Service

```
// Get tag list from the reader object obtained above using the
// client.getReader() method.
java.util.Collection c = reader.getTagList(readerName, 500);
if (c != null){
    Iterator i = c.iterator();
    while (i.hasNext()) {
        Event event = (Event)i.next();
        if (event instanceof IdentifierListEvent) {
            IdentifierListEvent eventList = (IdentifierListEvent) event;
            System.out.println("The following tags were read from the
             reader: "
                 + eventList.getSource());
            Iterator j = eventList.getTagList().iterator();
            while (j.hasNext()) {
                Identifier epc = (Identifier)j.next();
                System.out.println("tag= " + epc.getURI());
            }
     }
}
```

Implementation of the `PrinterClient` API

The implementation of the `PrinterClient` API is contained in the `com.sun.autoid.util.PrinterClient` class and is packaged in the `sun-rfid-common.jar` JAR file.

The `PrinterClient.java` class extends the `ReaderClient.java` class and offers additional methods for printing tags. A printer client can be used to find printers using the printer adapter name in the same manner that reader clients find readers using the `ReaderClient` APIs.

This section covers the following topics:

`PrinterClient` API Reference

All of the public `PrinterClient` APIs are listed in the following table.

TABLE 3-2 PrinterClient Interfaces

Constructors and Methods	Description
`PrinterClient(String printerName)`	Creates a new instance of `PrinterClient` matching on the printer name in ALL groups.
`PrinterClient(String printerName, String[] groups)`	Creates a new instance of `PrinterClient` matching on the printer name in groups specified by the groups parameter.

TABLE 3-2 PrinterClient Interfaces *(Continued)*

Constructors and Methods	Description
`PrinterClient(String printerName, String[] groups, LookupLocator[] locators)`	Creates a new instance of `PrinterClient` matching on the printer name in groups specified by the groups parameter. The local subnet is searched as well as any additional hosts that are specified by the `locators` parameter.
`printTag(Identifier id, Properties properties)`	Programs the single tag in the printer field of action and prints a label. The id is the Identifier that is written to the tag. The properties are applied to a template.
`printTag(Identifier id, String printStream)`	Programs the single tag in the printer field of action and prints a label. The id is the Identifier that is written to the tag. `printStream` is a `String` that is sent directly, unchanged, as a `print` command for this printer.

Building a Sample Printer Client

A sample printer client program is included in the Sun Java System RFID Software Toolkit. After unzipping the `RfidToolkit.zip` file into a directory of you choice, *toolkit-dir*, you can find `SamplePrinterClient.java` in the following directory: *toolkit-dir*`/samples/readerAccess/standAlone`.

Before running `SamplePrinterClient`, confirm the following prerequisites:

- The RFID Event Manager has been started.
- You have properly configured the printer adapter using the RFID Configuration Manager. Confirm that the configuration contains the correct IP address and port for the printer device.

▼ To Set Up the Sample Printer Client Environment

1. **Change to the directory containing the printer client sample program.**

   For example, on a Solaris system where *toolkit-dir* represents the directory where you downloaded and unzipped the RFID Software Toolkit zip file.

   ```
   cd toolkit-dir/samples/readerAccess/standAlone
   ```

2. **Verify that the `build.properties` file is correct for your installation.**

   You might need to modify the `rfid.home` property to point to your specific RFID Event Manager installation.

3. **Verify that the `build.xml` file target `sampleprinterclient` contains the correct `jvmarg` parameter values for your environment as follows.**

   - `-Dcom.sun.autoid.ReaderName` – The name of the printer to which you want to print. In the default `build.xml` file, the `ZebraPrinter` is specified.
   - `-Dcom.sun.autoid.JiniGroup` – The name of the Jini group that you specified during installation of your RFID Event Manager. The default value is AutoID-*hostname.*
   - `-Dcom.sun.autoid.TagBitSize` – The size of the tag that you want to write.

   `arg` – Specifies the identifier to print on the tag. Replace the `arg` value with the identifier that you want printed on the tag. The default value looks similar to the following:

   ```
   <arg value="urn:epc:id:gid:10.1002.37">
   ```

▼ To Run the Sample Printer Client Program

1. **Change to the directory containing the printer client sample program.**

   For example, on a Solaris system where *toolkit-dir* represents the directory where you downloaded and unzipped the RFID Software Toolkit zip file.

   ```
   cd toolkit-dir/samples/readerAccess/standAlone
   ```

2. **Run the printer client using the** `ant samplegprinterclient` **target.**

```
> ant sampleprinterclient
```

Explaining the Sample Printer Client

Configure a printer device using the RFID Configuration Manager. This example prints to a Zebra Technologies printer with the device name `ZebraPrinter`. See *Sun Java System RFID Software 3.0 Administration Guide* for procedures to define the physical printer device for the RFID Event Manager.

The default `build.xml` file specifies the `readerName` variable as `ZebraPrinter`. If you configure a different printer adapter, you need to modify this variable in the `sampleprinterclient` target. Modify the tag identifier that you wish to print by modifying the `arg` value parameter in the `sampleprinterclient` target of `build.xml` as described in Step 3 of the procedure "To Set Up the Sample Printer Client Environment" on page 70.

The first step in creating a printer client is to set the system security manager. This is necessary because the printer client uses Jini network technology to discover the printer.

See "Setting an RMI Security Manager and the Security Policy" on page 66.

Use one of the three `PrinterClient` constructors to find the printer.

CODE EXAMPLE 3-6 Finding the `ZebraPrinter` Printer Instance

```
String printerName = System.getProperty("com.sun.autoid.ReaderName",
                                "ZebraPrinter");
try {
      PrinterClient client = new PrinterClient(printerName);
      // Wait while looking for the reader
System.out.println("Wait while looking for the " + printerName
                                + " reader ...");
       Thread.sleep(3*1000);
} catch (Exception ex){
               ex.printStackTrace();
}
```

After creating the `PrinterClient` object, you can use it to print tags. The following code example shows how to print the `Identifier urn:epc:id:gid:10:1002:37`, which was specified in the default build.xml file, using the `ZebraPrinter` instance that was discovered using CODE EXAMPLE 3-6.

CODE EXAMPLE 3-7 Printing to the `ZebraPrinter` Instance

```
// Create properties for printing
 int count = 1; // default
 Properties properties = new Properties()
 properties.put("template", "default"); // default
 properties.put("COUNT", String.valueOf(count));
 properties.put("description", "Sample");
 properties.put("SHIP_TO_CUSTOMER_NAME",
                "Sun Microsystems”);
 properties.put("SHIP_TO_ADDRESS1", "Network Circle");
 properties.put("SHIP_TO_CITY", "Santa Clara");
 properties.put("SHIP_TO_STATE_PROV", "California");
 properties.put("SHIP_TO_POSTAL", "94087");

 String UCC = null;
 if (id instanceof EPC_SGTIN_BASE) {
     UCC = ((EPC_SGTIN_BASE)id).getGTIN14();
 }else if (id instanceof EPC_SSCC_BASE) {
     UCC = ((EPC_SSCC_BASE)id).getSSCC();
 }else if(id instanceof EPC_GIAI_BASE){
    UCC = ((EPC_GIAI_BASE)id).getGIAI();
 }else if (id instanceof EPC_GRAI_BASE) {
     UCC = ((EPC_GRAI_BASE)id).getGRAI();
 }else if (id instanceof EPC_SGLN_BASE) {
     UCC = ((EPC_SGLN_BASE)id).getSGLN();
 }
 if(UCC != null)
     properties.put("UCC", UCC);
 // Finally, print the Identifier
 .printTag(id, properties);
```

CHAPTER 4

Using Web Services for Device Access

In addition to the Java APIs described in Chapter 3, the Sun Java™ System RFID Software 3.0 provides two web services that can be used by Java and non-Java client applications to access RFID devices. The web services for device access can be installed by using the custom installation option of the RFID Event Manager installer. This chapter describes how to use these web services and includes the following topics:

Overview of Web Services for Device Access

These web services expose the device client Java interface of Sun RFID Event Manager. So any client application, including standalone Java applications, web clients, J2EE application clients, and other native clients, can use the common XML-based interface to interact with readers configured in the RFID Event Manager

The device access web services are only available if you install the RFID Event Manager using the custom installation option and choose the Web Services for ALE and Device Access component. See the *Sun Java System RFID Software 3.0 Installation Guide* for information on using this option of the RFID Event Manager installer.

The device access web services are described by the WSDL located at the following URLs:

- `http://`*em-hostname*`:`*app-server-port*`/readeraccess/ReaderAccess?WSDL`
- `http://`*em-hostname*`:`*app-server-port*`/printeraccess/PrinterAccess?WSDL`

The variable, *em-hostname*, is the host name of the machine where the RFID Event Manager is installed. The *app-server-port*, is your application server HTTP port number.

For example, using Sun Java System Application Server 8.1 Platform Edition listening on port 8080, the URL is `http://localhost:8080/readeraccess/ReaderAccess?WSDL`.

Web Services Interface Reference

This section describes the interfaces for the device access web services as follows:

- Web Services for Reader Access Java Interface
- Web Services for Printer Access Java Interface

The following terms are used in the interface descriptions:

- **Identifier** – Refers to the numeric ID stored in the RFID tag.
- **EPC** – Electronic Product Code. An identifier that follows the EPCglobal TDS 1.1 specification.
- **TagType** – The type of RFID tag. The type can be ISO 18000-6B, EPCglobal, UHF Gen2, EPCglobal Class 1, EPCglobal Class 0 or 0+, and others. See the API specifications for the `com.sun.autoid.identity` package. The API specifications are included with the RFID Software Toolkit – *toolkit-dir*`\docs\api\index.html`
- **Indicators** – External devices that can be connected to the RFID reader, such as industrial light stacks (red/yellow/green). The lights can indicate if a tag has been successfully read. The indicators might control whether a process should proceed or be halted for inspection.

Web Services for Reader Access Java Interface

The implementation of the reader access Java interface API is contained in the following class, `ReaderAccessSEI.java`.

The following table lists the public `ReaderAccessSEI` APIs:

TABLE 4-1 `ReaderAccessSEI` Interfaces

Method	Description
`runCommand`	Sends a command to a RFID reader.
`getStatus`	Returns the status of the RFID reader.
`triggerTagList`	Tells all the RFID readers to inquire about the list of tags in the readers field of view.
`getTagList`	Gets the current tags from the RFID reader.
`writeIdentifier`	Write an `Identifier` to the RFID reader.
`changeIndicators`	Turns the indicators attached to the RFID reader on or off.
`readUserData`	Reads the contents of the user data memory area for the identified RFID tag.
`writeUserData`	Writes the specified data to the user data memory area of the specified RFID tag.
`getUserDataSize`	Returns the size of the user data memory area in bytes.
`getTagProtocol`	Returns the protocol for the specified RFID tag.
`getTagType`	Returns the type of the specified RFID tag.
`lockEpc`	Some RFID tags contain an Identifier memory area that is read-only. If the tab memory area supports read/write capability, this method can be used to lock the EPC identifier area of the RFID tag.
`lockUserData`	Locks the user memory data area of the specified RFID tag.
`killTag`	Permanently disables the RFID tag for the specified `Identifier`.
`gatherUserDataEnabled`	Determines if automatic gathering of the user data is necessary during the RFID tag inventory operation.
`enableGatherUserData`	Enables or disables the automatic gathering of user data during the RFID tag inventory operation.
`getEPC`	Get the reader EPC identifier from the reader name.
`getDescription`	Get the reader description.
`getHandle`	Get the reader `Handle`.

TABLE 4-1 `ReaderAccessSEI` Interfaces *(Continued)*

Method	Description
`getRole`	Get the reader `role`. • This field is used to indicate a business process role and should not be confused with the concept of an RFID Event Manager role, a term that is described in the *Sun Java System RFID Software 3.0 Administration Guide*.
`isAutoRead`	Returns true if automatic read mode is on. Returns false if otherwise.
`setAutoRead`	Enables or disables the automatic read mode.

Web Services for Printer Access Java Interface

The implementation of the printer access Java interface API is contained in the following class, `PrinterAccessSEI.java`.

All of the public `PrinterAccessSEI` APIs are listed in the following table.

TABLE 4-2 PrinterAccessSEI Interfaces

Method	Description
`getStatus`	Returns the status of the RFID printer.
`getEPC`	Returns the RFID printer EPC `Identifier` from printer name.
`getDescription`	Returns the RFID printer description.
`getHandle`	Returns the RFID printer `Handle`.

TABLE 4-2 PrinterAccessSEI Interfaces *(Continued)*

Method	Description
`getRole`	Returns the RFID printer `Role`.
`PrintTag` (with properties)	• Programs the single tag in the RFID printer's field of action and prints a label. The `Properties` object is applied to a template identified by the `template` property. The default template is used, if a `template` property is not specified. • When you invoke the `printTag` method, the framework automatically adds the values for the properties `HEXEPC` and `URNEPC` to the `Properties` object. You still need to include the two properties in your template. • `Strings` of the form `${rfid.`*`myproperty`*`}` in the template are replaced with the value of *myproperty* from the Properties object that is passed to the `printTag` method. This mechanism enables each printed label to have data customized for that specific label.
`PrintTag` (with data stream)	Programs the single tag in the RFID printer's field of action and prints the label with the data contained in the supplied `String`. The `String` is sent to the printer verbatim with no dynamic substitution of data.

Creating and Running the Web Services for a Device Access Client

To discover a specific reader service in the RFID Jini lookup server, you need to supply the following parameters:

- `String name` – The reader or printer name
- `String groups` – The Jini lookup group names
- `String locators` – The Jini lookup locators

These parameters are described in more detail in "Reader Client Constructor Parameters" on page 54 of Chapter 3 of this guide.

To simplify the call parameters, the `DeviceFinder` class has been defined to package these three parameters. You must create a `DeviceFinder` object that defines the name of the reader you wish to access, along with the necessary Jini information, such as the list of Jini groups in which to search for the reader and any Jini locators that may be necessary to find readers running in an RFID Event Manager on a different subnet. The list of Jini groups and Jini locators may be null. If null is specified for the `groups` argument, then all Jini groups are searched. If null is specified for the `locators` argument, then only readers running on the same

subnet on which the client is running are found. You can specify multiple groups and locators in either `String` by using a space, tab, or comma as the separator character.

The four types of device clients that can be used to interact with the RFID device access web services are the following:

- Static stub client
- Dynamic proxy client
- Dynamic invocation interface client
- Application client

This guide focuses on the first two methods and provides examples of how to develop the reader access client. The examples are packaged as NetBeans projects and are included in the Sun Java System RFID Software Toolkit (RFID Software Toolkit).

Prerequisites for Running the Web Services Client Examples

To use the examples you need an installation of the NetBeans 4.1 IDE. You also need to download and install the RFID Software Toolkit `.nbm` and set up your RFID environment. Use the following procedures to set up your NetBeans environment.

- "To Download and Install NetBeans" on page 22
- "To Download and Install the RFID Software Toolkit" on page 22

If you are not familiar with the NetBeans 4.1 IDE, refer to the NetBeans 4.1 Quick Start Guides.

▼ (Optional) To Access the NetBeans IDE 4.1 Quick Start Guide for Web Services

1. **After installing and starting the NetBeans 4.1 IDE, choose Help → Tutorials → Quick Start Guide.**

   The Getting Started with NetBeans IDE 4.1 Quick Start Guide web page appears in your web browser.

2. **Click Web services.**

   The NetBeans IDE 4.1 Quick Start Guide for Web Services web page appears.

3. **Review the instructions as needed to get started creating your own web services clients.**

   When you are ready, proceed to the examples in this chapter. The following sections of this chapter describe using the `ReaderAccess` client example to illustrate how to develop a web service client.

▼ To Configure the Environment for the Web Services Client Examples

Prerequisite – Confirm that you have installed the NetBeans 4.1 IDE and installed the RFID Software Toolkit. See "Prerequisites for Running the Web Services Client Examples" on page 78.

1. **Start your application server.**
2. **Confirm that the reader client web service has been deployed as part of your RFID Event Manager installation.**

   See the *Sun Java System RFID Software 3.0 Installation Guide*.
3. **Confirm that Sun Java System Application Server 8.1 is configured in NetBeans.**

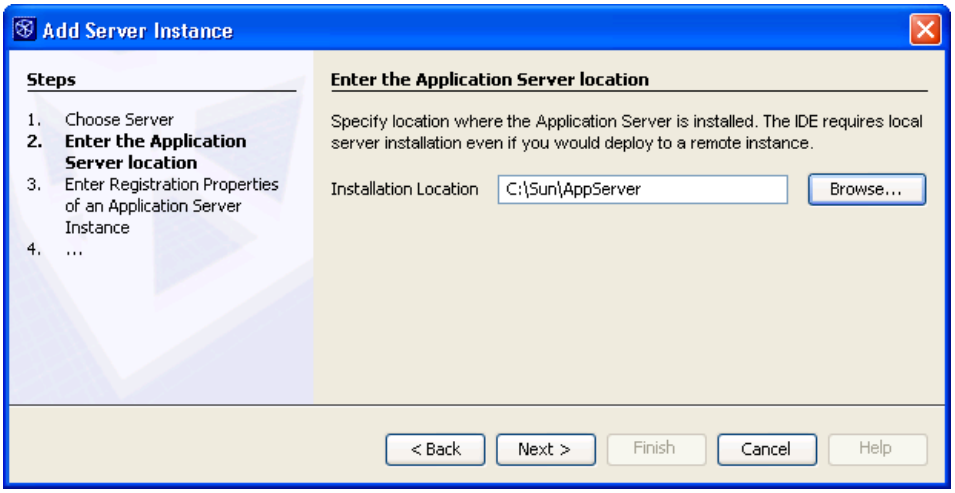
   For example, use the following steps:

   a. **In the IDE's Runtime window, right-click Servers and choose Add Server from the contextual menu.**

      The Choose Server pane of the Add Server wizard appears.

   b. **Select Sun Java System Application Server 8.1 and click Next.**

      Browse to the location of your application server installation.



c. Select your application server and click Next.

The Enter Registration Properties pane of the wizard appears.

d. Type the registration properties following the instructions on the wizard pane and click Finish.

The Sun Java System Application Server 8.1 instance is added to the IDE's Runtime window.

4. Start the RFID Event Manager and the readers that you plan to use.

For this example, start the PMLReader.

Writing the Static Web Services Client

The static client example shows the simplest way to write a client for the reader access web service. The static web service client makes method calls through a stub, a local object that acts as a proxy for the remote service. Because the stub is created at development time (as opposed to runtime), it is usually called a static stub.

▼ To Run the Static Web Services Client Example

1. Confirm that you have successfully complete the procedure, "To Configure the Environment for the Web Services Client Examples" on page 79.

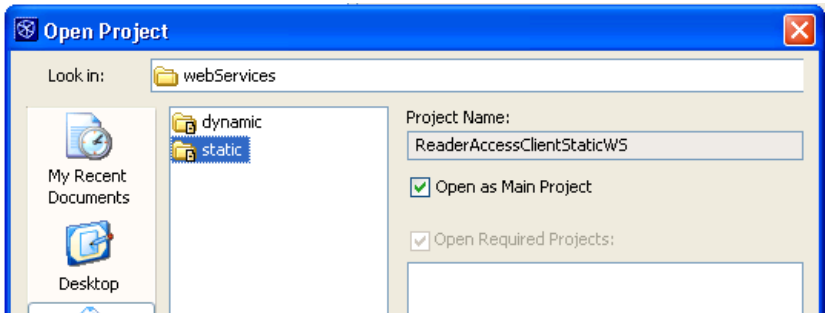
2. Start the NetBeans 4.1 IDE.

3. **Choose File → Open Project Folder.**

   The Open Project dialog box appears.

4. **Navigate to the directory containing the static client.**

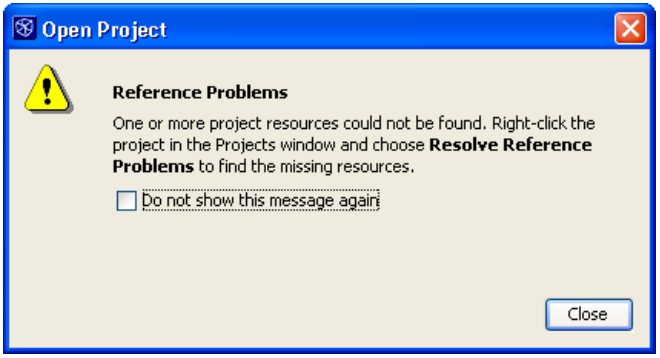
   For example, if the directory where you downloaded and unzipped the RFID Software Toolkit is *toolkit-dir*. The static client is located at *toolkit-dir*`/samples/readerAccess/webServices/static`. The `ReaderAccessClientStaticWS` project is listed as shown in the following screen capture.



5. **Click Open Project Folder.**

   To resolve the reference problem described in the following message, create a library named `jwsdp`. See "To Create the RFID Library for the Custom Component Examples" on page 28 for the general steps to use. Add the JAR files in the following locations:

   - UNIX – /usr/share/lib
   - Microsoft Windows – `C:\Program Files\Sun\RFID Software\rfidem\lib\utils`



6. **In the Projects window, navigate to the following files and configure the URL with the correct host name and port number.**

   http://*em-hostname*:*app-server-port*/readeraccess/ReaderAccess

   - ReaderAccessClientStaticWS → Source Packages → com.mcompany → ReaderAccessClient.java
   - ReaderAccessClientStaticWS → Source Packages → conf → ReaderAccess-client-config.xml

7. **In the Projects windows, right-click the static client project node and choose Run Project.**

   The IDE rebuilds and runs the project.

Writing the Dynamic Web Services Client Example

In contrast to the static web service client described in the preceding section, a dynamic client calls a remote procedure through a dynamic proxy. The dynamic proxy is a class that is created during runtime. Although the source code for the static stub client relies on an implementation-specific class, the code for the dynamic proxy client does not have this limitation.

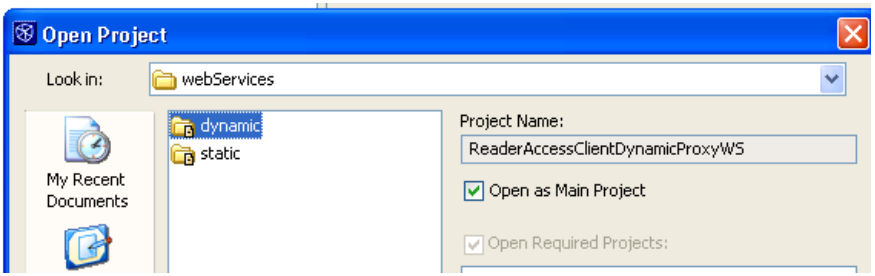
▼ To Run the Dynamic Web Services Client

1. **Confirm that you have successfully complete the procedure, "To Configure the Environment for the Web Services Client Examples" on page 79.**
2. **Start the NetBeans 4.1 IDE.**
3. **Choose File → Open Project Folder.**

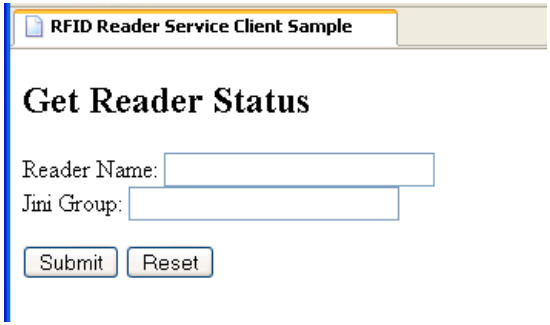
   The Open Project dialog box appears.

4. **Navigate to the directory containing the dynamic web service client.**

   For example, if the directory where you downloaded and unzipped the RFID Software Toolkit is *toolkit-dir*. The dynamic client is located at *toolkit-dir*`/samples/readerAccess/webServices/dynamic`. The `ReaderAccessClientDynamicProxyWS` project is listed as shown in the following screen capture.

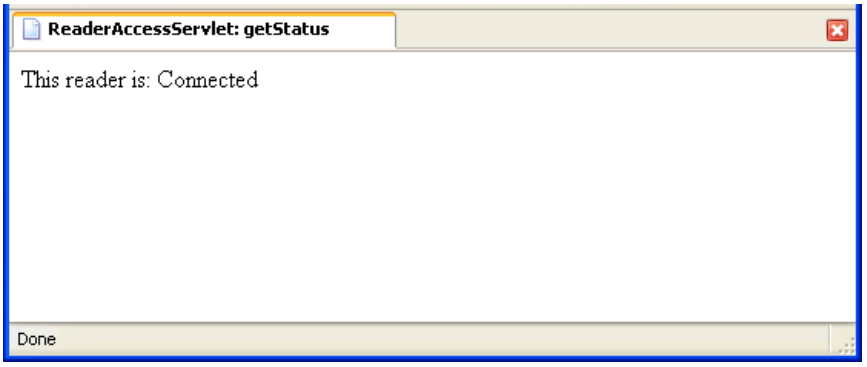


5. **Click Open Project.**
6. **(Optional) To refresh the WSDL, use the following steps.**
   a. **In the Projects window, expand the Web Service References node.**
   b. **Select the ReaderAccess.wsdl node and right-click.**
   c. **Choose Refresh WSDL from the contextual menu.**
      The Refresh WSDL for Web Service Client dialog box appears.
   d. **Change the Original WSDL location if necessary and click OK.**
7. **In the Projects window, navigate to the following files and configure the URL with the correct host name and port number.**
   http://*em-hostname*:*app-server-port*/readeraccess/ReaderAccess
   - Web Pages → WEB-INF → wsdl → ReaderAccess-config.xml
   - Configuration Files → ReaderAccess-client-config.xml
8. **Right-click the project node, ReaderAccessClientDynamicProxyWS, and choose Run Project from the contextual menu.**
   The IDE rebuilds the project and deploys it to your application server. The ReaderAccessClient servlet is invoked and a JSP appears in your web browser as shown in the following screen capture.



9. **Type the Reader Name and the Jini Group and click Submit.**

   You see something similar to the following in your web browser.



CHAPTER 5

ALE Web Services

This chapter describes the Sun Java System RFID Software 3.0 implementation of the EPCglobal Application Level Events (ALE) specification, also known as collection and filtering. ALE is a web service specification containing a WSDL to define, configure, and request reports. There is also an XML schema for requesting reports and for the reports themselves. You should be thoroughly familiar with *Application Level Events (ALE) Specification, Version 1.0* before using the ALE components. The specification can be found at the EPCglobal web site `http://www.epcglobalinc.com.`

The following topics are included in this chapter:

Broad Architecture

The Sun Java RFID Software implements the ALE web service using Java API for XML-based RPC (JAX-RPC). This is contained in the `sun-ale-service.war` file and is deployed to an application server, such as Sun Java System Application Server 7, 8.1. This service acts as an intermediary to a Jini RMI service contained within the RFID Event Manager.

The implementation of ALE in the RFID Event Manager is based on a new service named ALE. This service implements the WSDL methods described in the specification. The ALE service uses Java RMI (Java Remote Method Invocation). Report requests and report messages are implemented as POJO (Plain old Java objects) in the package `com.sun.autoid.ale.spec`.

To conform to the ALE specification, the ALE WSDL is processed using JAX-RPC to generate Java web service classes. These classes include client-side classes that work with the POJO objects contained in `com.sun.autoid.ale.spec`. The ALE WSDL common server and client code is packaged into `sun-alesvc-common.jar`. An ALE client API is implemented to hide some of the complexity of the JAX-RPC code. The ALE client API is described in the section titled, "Using ALE Web Services Client API". A sample of using the ALE client API is listed in CODE EXAMPLE 4-2.

The ALE XML schema is processed using JAXB and the generated files are placed into package `com.sun.autoid.ale.xml`. There is generally a one-to-one mapping of the JAXB generated objects to the classes in `com.sun.autoid.ale.spec`. The class `com.sun.autoid.ale.XMLUtil` provides methods to translate back and forth between the JAXB representation and the POJO representation of the spec classes.

ALE functionality includes a requirement to register for reader events based on the ALE request. So, the ALE service discovers all readers and when a request becomes active, the ALE service registers with the event producer (`com.sun.autoid.logger.REProducer` in Connector) on the reader. The mechanics of how this is done is described later.

ALE Service Architecture

The ALE Service is deployed as a Jini RMI service in the RFID Event Manager. Each ALE report request is translated into an event processing network, including devices, filters, and connectors.

The ALE specification contains the concept of a physical reader and a logical reader. A logical reader can be comprised of one or more physical readers. To discover the physical or logical readers, the ALE implementation uses the reader name as defined in the Configuration Object for the reader and also uses the `logicalReaders` property for the reader. The `logicalReaders` property comprises a comma-separated list of logical names. A reader may belong to more than one logical group. For example, if a reader belongs to the logical groups named Dock Door 1 and Receiving, the `logicalReaders` property is set to the values Dock Door 1 and Receiving.

The ALE implementation constantly searches for all readers on the system and maintains the current list of physical and logical names. If the reader service goes offline and the reader specified in the ALE event cycle specification cannot be located, an ALE exception is thrown.

The `ALEEventFilter` implements the event cycle as described in the ALE specification. The event cycle is a state machine as described in the specification. Basically, an event cycle is started, tag events are gathered, and when the event cycle finishes, the tag list is pushed out to the `ALEEventReportFilter`. To fully understand the `ALEEventFilter` it is necessary to understand the ALE specification. An `ECSpec` describes an event cycle and one or more reports that are to be generated from it. It contains a list of logical Readers whose read cycles are to be included in the event cycle, a specification of how the boundaries of event cycles are to be determined, and a list of specifications each of which describes a report to be generated from this event cycle. A sample of an ECSpec is listed in CODE EXAMPLE 5-1.

The ALE specification identifies external triggers that can start or stop the event cycle. These triggers are specified as URIs, but the interpretation of the URI is up to the implementation. In Sun's implementation, the URIs behave as follows:

- An EPC pattern-matching URI - the trigger is fired when an EPC is seen that matches.
- JMS URI - designates a queue or topic to receive messages
- HTTP URI - which is opened and read to receive a message.
- FILE URI - which is opened and read to receive a message.
- TCP URI - designates a socket, which is opened and read to receive a message.
- Anything else, default is to always fire.

For the JMS, HTTP, FILE and TCP URIs the format of the payload is a simple XML document. For example:

```
<Trigger fired="true" />
<Trigger fired="false"/>
```

At the end of an event cycle, as defined by the ALE specification, the `ALEEventFilter` creates an `ALEEvent` that contains a list of tags and a list or readers that produced the tag events. This is passed to the `ALEEventReportFilter` which takes the input and creates a `MiscEvent` that contains a property that holds the `ECReport` object.

In the ALE specification, the definition of the report and the subscription to receive the report output are two distinct operations. Therefore, when a subscribe request comes in, a specialized logger is created to handle the request.

The types of loggers are:

- JMS Logger - the Report XML is include as the payload message to a JMS queue or topic
- File logger - where the Report XML is written to a local file
- TCP logger - where the Report is written to a socket

- Poll logger - where the Report is returned as a String to the poll request
- All other loggers are treated as an URL, which is opened for writing

An event cycle cannot start until at least one subscriber is registered to receive the reports. When subscriptions are registered, the appropriate `Logger` is created and dynamically linked to the `ALEEventReportFilter`. Similarly, when a subscription is removed, the `Logger` is stopped and disconnected from the `ALEEventReportFilter`.

Lastly, when a Report specification is "undefined," all the processing components are stopped, and dynamically removed from the ALE service.

Other Considerations

Because ALE has been implemented as RFID Event Manager components, the components can be used in the same way other components are used. The ECSpec can be set up using the component properties. In this way, the ALE components can be permanently attached to a reader, if needed. A sample configuration file demonstrating this can be found in the *Sun Java System RFID Software 3.0 Administration Guide*, Appendix C.

CODE EXAMPLE 5-1 Sample ECSpec file

```
<?xml version="1.0" encoding="UTF-8"?>
<ECSpec xmlns="urn:epcglobal:ale:xsd:1"
        includeSpecInReports="true"
        creationDate="2005-02-07T13:42:40.790-05:00"
        schemaVersion="1.0">
   <logicalReaders xmlns="">
     <logicalReader>Reader</logicalReader>
   </logicalReaders>
   <boundarySpec xmlns="">
     <startTrigger>http://localhost/start</startTrigger>
     <duration>2000</duration>
     <stableSetInterval>0</stableSetInterval>
   </boundarySpec>
   <reportSpecs xmlns="”">
     <reportSpec reportIfEmpty="false”
           reportName="report"
           reportOnlyOnChange="false”>
       <reportSet set="CURRENT"/>
       <output includeCount="false” includeList="true"/>
     </reportSpec>
   </reportSpecs>
</ECSpec>
```

Using ALE Web Services Client (ALEClient) API

The ALE web service client, `MyAleClient.java`, is a sample program for the purposes of illustrating how to create a Java client that communicates with the ALE web service. It is installed when the ALE custom option is selected during installation of the RFID Event Manager and is located in *rfid-install-dir*`/rfidem/samples/ale/aleclient`. Where *rfid-install-dir* is one of the following depending on your platform:

- Solaris – `/opt/SUNWrfid`
- Linux – `/opt/sun/rfidem`
- Microsoft Windows – `C:\Program Files\Sun\RFID Software\rfidem\`

This sample includes all the necessary tools for compiling and running the sample program. It also includes a sample ECSpec that can be used to query the ALE web service for tag data.

Client Checklist

Before running `MyAleClient`, be sure to confirm the following items:

- The ALE client software must be started on the same machine on which the RFID Event Manager is installed. This is necessary because the ALE service depends on specific JAX libraries, which are installed with the RFID Event Manager. Alternatively, you can copy the necessary files from a machine where the RFID Event Manager is already installed and then set the `JAX_LIB_PATH` environment variable to that directory. See Step 3 in the procedure "To Set Up the ALE Client Environment" on page 90.
- Your application server is running
- Confirm that the `server.policy` file in the application server has been updated by the Event Manager installer or you did it manually.

Note – Note, the application server must be restarted after the policy file has been changed.

- The ALE web service was properly deployed to the application server. See Chapter 3 – "To Install the Event Manager Using Custom Installation" of the RFID Software Installation Guide for instructions on installing the ALE web service.
- The RFID Event Manager is running.
- Confirm that you are receiving tags from the Event Manager in the tag viewer

▼ To Set Up the ALE Client Environment

1. **Change to the directory containing the ALE sample directory.**
   - Solaris - `/opt/SUNWrfid/samples/ale/aleclient`
   - Linux - `/opt/sun/rfidem/samples/ale/aleclient`
   - Windows - `C:\Program Files\Sun\RFID Software\rfidem\samples\ale\aleclient`
2. **Create a `lib` directory in the `aleclient` directory.**

   Solaris - mkdir -p lib
3. **Copy the following `JAR` files to the new `lib` directory.**
   - Solaris - `/opt/SUNWrfid/lib/sun-alesvc-common.jar and sun-rfid-common.jar`
   - Linux - `/opt/sun/rfidem/lib/sun-alesvc-common.jar and sun-rfid-common.jar`
   - Windows - `C:\Program Files\Sun\RFID Software\rfidem\lib\sun-alesvc-common.jar` and `sun-rfid-common.jar`
4. **Verify that the `build_properties.xml` file is correct for your installation.**

▼ To Run the ALE Web Services Client

1. **Be sure that you have confirmed all items in the client checklist (see "Client Checklist" on page 89) and performed the procedure "To Set Up the ALE Client Environment" on page 90.**
2. **Set the environment variable, *JAX_LIB_PATH* as follows. For example, using the `csh` shell for Solaris and Linux:**
   - Solaris

```
setenv JAX_LIB_PATH /usr/share/lib
```

   - Linux

```
setenv JAX_LIB_PATH /opt/sun/share/lib
```

   - Windows

```
set JAX_LIB_PATH=\C:\Program Files\Sun\RFID Software\rfidem\\lib\utils
```

3. **Edit the `build.xml` file found in the sample aleclient directory and change the *localhost* and port number to match your target environment.**

   For example, change the `runMyAleClient` ant target value as follows – change the following:

```
<arg value=http://localhost/ALEService/ale
```

   to the following:

```
<arg value=http://em-hostname/app-server-port/ALEService/ale
```

Troubleshooting for ALE Client

Symptom: You see an `ImplementationException` with the following message when you run the ALE client

```
[java] Message for the exception: Connection to ALE Services is
not available at this time
```

Solution: Confirm that the RFID Event Manager is running and sufficient time has elapsed to enable all components to start.

CHAPTER 6

Using RFID Information Server Client API

This chapter describes the Sun Java™ System RFID Software Information Server and the client API for interfacing with RFID Information Server.

The following topics are covered:

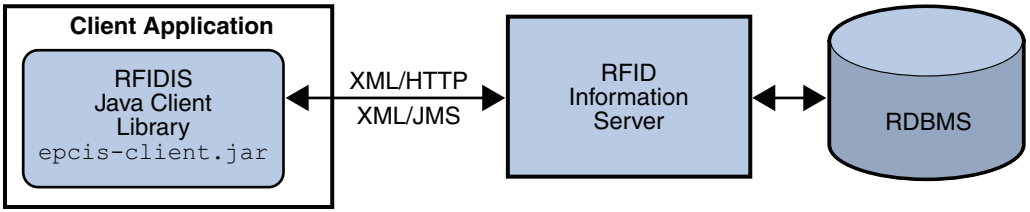
Architecture

The Sun Java System RFID Software Information Server (IS) is a J2EE application that runs on one of the supported application servers. RFID Information Server stores all data in a relational database. In this release of Sun Java System RFID Software, RFID Information Server is supported on the following databases:

- Oracle Database 9i
- Oracle Database 10g
- PostgreSQL 8.0.4

External applications interface with the RFID Information Server through XML message exchange. Requests and responses are expressed in XML and conform to an XML schema. The RFID Information Server supports HTTP and JMS message transports. The RFID Software provides a Java client library that can be used to

access the RFID Information Server from your software applications. The APIs used to query and manipulate data in the RFID Information Server are independent of the protocol used. The following figure shows the architecture.



Database Tables

This section describes the Sun Java System RFID Software database table names and keys.

TABLE 6-1 RFID Information Server Database Tables

Name	Definition	Keys
CONTAINER_TYPE	Each entry in this table represents a type of container. Container types are specific to a deployment. Common container types include pallets, cases, inner packs and items.	Primary – NAME
CONTAINMENT	Maintains a hierarchy of container relationships between EPCs. Each container is identified by the parent EPC. The parent may have zero or more child EPCs. Because a child EPC can represent a container, the hierarchy can be arbitrarily deep.	Primary – EPC
CONTAINMENT_LOG	Maintains the history of containment relationships between container EPCs.	

TABLE 6-1 RFID Information Server Database Tables *(Continued)*

Name	Definition	Keys
CURRENT_OBSERVATION	Maintains the list of tags that are being reported as currently visible at a sensor. This table can only be used in conjunction with the RFID Event Manager's Delta filter. To keep the table consistent the RFID Information Server relies on the `TagsIn` and `TagsOut` property of the Delta event.	Foreign – SENSOR_EPC
CUSTOMER	Maintains the customer shipment information.	Primary – CUSTOMER_ID
EPCLOG	Maintains a history of shipments. This log provides a record of the EPCs (each representing a specific product or container) that comprised each shipment.	Primary – LOG_ID Foreign – EPC, SHIPPING_ID
LOCATION	Maintains the physical locations.	Primary – LOCATION_ID Foreign – EPC
OBSERVATION_LOG	Maintains the history of all tag observations. Each entry in the table represents an observation which contains the EPC of the observer (sensor), the EPC of the observed value and the time at which it was observed.	Foreign – SENSOR_EPC
ORGANIZATION	A manufacturer or distributor of goods.	Primary – ORGANIZATION_ID
ORGANIZATION_EXT	Maintains a set of properties associated with an ORGANIZATION entry. These properties reference their respective ORGANIZATION using the ORGANIZATION_ID field. The properties consist of name and value pairs.	Foreign – ORGANIZATION_ID
ORGANIZATION_XREF	Identifies the hierarchy of an organization. This table is designed to describe a structure where an organization is part of a division and the division is part of a company and so on.	Primary – ORGANIZATION_ID Foreign – ORGANIZATION_ID, PARENT_ORG

TABLE 6-1 RFID Information Server Database Tables *(Continued)*

Name	Definition	Keys
PRODUCT	A class of items, or Stock Keeping Unit (SKU) identified by a GTIN or UPC code.	Primary – PRODUCT_ID Foreign – MANUFACTURER_ID
PRODUCT_EXT	Maintains a set of properties associated with a PRODUCT entry. These properties reference their respective PRODUCT instance using the PRODUCT_ID field. The properties consist of name and value pairs.	Foreign – PRODUCT_ID
SENSOR	An RFID reader or antenna uniquely identified by an EPC.	Primary – EPC
SHIPPING_INFO	Maintains the information specific to a particular shipment. By using this table, the CUSTOMER table, and EPCLOG, it is possible to find all of the shipments to a specific customer and to identify which products or containers were in the shipments.	Primary – SHIPPING_ID Foreign – CUSTOMER_ID
TAG_ALLOCATION	Assigns and de-assigns ranges of EPC number and keep track of them.	Primary – TAG_ALLOCATION_ID
TAG_ALLOCATION_LOG	Maintains the history of all tag allocated.	
TX_LOG	Associates a set of EPCs with a business transaction ID. Common transaction IDs include PO numbers and ASNs.	Foreign – UNIT
UNIT	A tagged entity identified by a unique EPC. An entry in the UNIT table can represent a pallet, case or any other entity that's tracked.	Primary – EPC Foreign – PRODUCT_ID, UNIT_TYPE, LOCATION_ID, OWNER_ID
UNIT_EXT	Maintains a set of properties associated with a UNIT entry. These properties reference their respective UNIT instance by using the EPC field. The properties consist of name and value pairs.	Foreign – EPC

Connecting to RFID Information Server

The `com.sun.autoid.epcis.client.EpcisConnection` class represents a communication link between an application and RFID Information Server. An instance of the `EpcisConnection` class can support multiple requests. The transport protocol used by the connection is specified using the appropriate constructor. The user name and password that are used are the ones that are specified for access to the RFID Information Server. The process of creating users for access to the RFID Information Server reports and index page is described in the installation guide for this product. See Chapter 5 in the *Sun Java System RFID Software 3.0 Installation Guide*.

You specify the RFID Information Server database schema by using one of the following Java System properties:

- `rfidis.db.schema` – Use this property to specify the URL of the database schema for the RFID IS.

  The format of the URL is `http://`*epcis-host*`:`*port*`/epcis/EpcisDbSchema.xml`. Replace the variable, *epcis-host*, with the host name or IP address where your RFID Information Server is installed. The variable, *port*, is an optional port number for Application Server. The usual default port number is `80`. For example, `http://host1.sun.com/epcis/EpcisDbSchema.xml` would point to the web server listening on port 80 that is hosting the RFID Information Server. The URL `http://host1.sun.com:8080/epcis/EpcisDbSchema.xml` specifies the web server on port 8080 that is hosting the RFID Information Server.
- `rfidis.db.schema.file` – Use this property to specify a file on the local file system that defines the database schema. For example `/opt/sun/schema/EpcisDbSchema.xml`.

You can set these properties in the following ways:

- On the command line when invoking java – For example, use the following command.

```
java -Drfidis.db.schema.file=/opt/sun/schema/EpcisDbSchema.xml
com.mycompany.rfid.ISClient
```

- From a software program – For example, use the following code.

```
System.setProperty("rfidis.db.schema.file",
"/opt/sun/schema/EpcisDbSchema.xml");
```

If at least one of these properties is not set, then the RFID Information Server client API tries to infer the value. If the `EPCISConnection` is an HTTP type, then the URL is constructed from the URL for the RFID Information Server that is passed in the `EPCISConnection` constructor. If this constructed URL can be successfully contacted, then the system property `rfidis.db.schema` is set to this constructed URL. If connecting to that URL fails, the following URLs are tested in order, `http://localhost/epcis/EpcisDbSchema.xml` and `http://localhost:8080/epcis/EpcisDbSchema.xml`. If connectivity is successful to one of these constructed URLs, then the property, `rfidis.db.schema` is set to the first one that successfully connects.

If all the connection attempts that use the constructed URLs fail, then an `EPCISException` is thrown from the `EPCISConnection` constructor.

If the properties, `rfidis.db.schema` and `rfidis.db.schema.file`, are set prior to the `EPCISConnection` constructor, then there is no connectivity check during the `EPCISConnection` constructor and no exception will be thrown. However, if connectivity cannot be established later on, when you access the RFID Information Server APIs, then `EPCISExceptions` are thrown at that time.

This section contains the following code examples:

- "Establishing a Connection Using HTTP" on page 99.
- "Establishing a JMS Topic Connection on Sun Java System Application Server 8.1" on page 99 – When using the File System Context include `fscontext.jar` in your class path. This JAR file can be found in the Application Server installation at *appsvr-install-dir*`/imq/lib`. Also see the *Sun Java System RFID Software 3.0 Administration Guide* for detailed instructions on enabling JMS usage.
- "Establishing a JMS Queue Connection on Sun Java System Application Server 8.1" on page 99 – The Java Naming and Directory Interface (JNDI) is part of the Java platform, providing applications based on Java technology with a unified interface to multiple naming and directory services. To enable the Application Server remote JNDI, you must first deploy the Message Queue Resource Adapter (`imqjmsra.rar`) to the Application Server. For a development reference, see the "Developing Java Clients" chapter in the *Sun Java System Application Server Platform Edition 8.1 2005Q1 Developer's Guide* at `http://docs.sun.com/source/819-0079/index.html`. Include the following JAR files in the class path:
  - `appserv-rt.jar` – available at *appsvr-install-dir*`/lib`
  - `j2ee.jar` – available at *appsvr-install-dir*`/lib`
  - `appserv-admin.jar` – available at *appsvr-install-dir*`/lib`
  - `imqjmsra.jar` – available at *appsvr-install-dir*`/lib/install/applications/jmsra`

Note – Do not include the `fscontext.jar` in your class path when using a remote JNDI.

- "Establishing a Connection Using JMS on BEA WebLogic Server 8.1 SP4" on page 100.
- "Closing the Connection" on page 100.

CODE EXAMPLE 6-1 Establishing a Connection Using HTTP

```
EpcisConnection conn = new EpcisConnection(
"http://hostname.xyz.com/epcis/service", // url to the IS service
 "proxy.xyz.com", // proxy host
 "8080",         // proxy port
 "myname",       // user name
 "mypassword"); // password
```

CODE EXAMPLE 6-2 Establishing a JMS Topic Connection on Sun Java System Application Server 8.1

```
EpcisConnection conn = new EpcisConnection(
"com.sun.jndi.fscontext.RefFSContextFactory",
"file:///imq_admin_objects", // file system JNDI provider URL
"TopicConnectionFactory", // name of the connection factory
"epcisTopic", // name of the topic
"true",      // user JMS (true or false)
"myname",    // user name
"mypassword");  // password
```

CODE EXAMPLE 6-3 Establishing a JMS Queue Connection on Sun Java System Application Server 8.1

```
EcpisConnection conn = new EpcisConnection (
null,//Initial context factory
"iiop://localhost:3700", //URL of the App Server JNDI provider
"jms/QueueConnectionFactory", //name of the Connection Factory for the JMS
queue
"jms/epcisQueue", //name of the domain for the JMS queue
EPCISConstants.CONNECTION_TYPE_JMS_QUEUE, //the domain of the JMS queue
"myname",    // user name
"mypassword");  // password
```

CODE EXAMPLE 6-4 Establishing a Connection Using JMS on BEA WebLogic Server 8.1 SP4[1]

```
EcpisConnection conn = new EpcisConnection (
"weblogic.jndi.WLInitialContextFactory", // JNDI initial context factory
"t3://localhost:7001", // JNDI provider URL
"jms/TopicConnectionFactory", // name of the connection factory
"jms/epcisTopic", // name of the topic
"true",
"username", // authentication username
"password"); // authentication password
```

1 If the initial-context-factory is not specified, the default is com.sun.jndi.fscontext.RefFSContextFactory.

Closing the Connection

Calling the `close()` method on the connection calls the `close()` method of the underlying `HttpURLConnection` or `TopicConnection` object. It is recommended that the call to the `close()` method be captured in a `finally` block.

CODE EXAMPLE 6-5 Closing the Connection

```
EpcisConnection conn = null;
try {
          conn = ... //initialize the connection here
} catch (Exception e) {
          e.printStackTrace();
} finally {
          try {
               conn.close();
          } catch (Exception e) {
                    e.printStackTrace();
          }
}
```

Exchanging Data With RFID Information Server

`Transfer` objects are client-side representations of data that are exchanged between RFID Information Server and an application. The data in a `Transfer` object may be stored in one or more database tables. The Java class to database object relationships are shown in TABLE 6-2. `Unit`, `Product`, `Organization`, `Container`, `Sensor`, `ContainerType`, `Transaction` and `Observation` are all transfer objects.

You would typically expect to be able to insert, query, modify and delete these objects from the database. But the `Transaction` and `Observation` objects are exceptions to this rule. You use the `UpdateRequest` object to record an `Observation` or `Transaction`. Once an `Observation` or `Transaction` is recorded, it cannot be modified. In addition, an `Observation` cannot be deleted. There is no notion of erasing a past observation. The OBSERVATION_LOG table maintains the history of observations.

TABLE 6-2 Java Class to Database Objects Relationships

Package Name	Java Class Name	Request Object	Database Object
`com.sun.autoid.epcis.client`	`ValueObjectWrapper` The superclass of all other classes in this table.	`Request`	
	`ContainerType`	`ContainerTypeRequest`	An entry in the CONTAINER_TYPE table
	`ContainmentLog`	`ContainmentLogRequest`	An entry in the `CONTAINMENTLOG` table
	`Customer`	`CustomerRequest`	An entry in the CUSTOMER table
	`EpcLog`	`EpcLogRequest`	An entry in the EPCLOG table
	`Location`	`LocationRequest`	An entry in the LOCATION table
	`Observation`	`ObservationRequest`	An entry in the OBSERVATION_LOG or CURRENT_OBSERVATION table

TABLE 6-2 Java Class to Database Objects Relationships *(Continued)*

Package Name	Java Class Name	Request Object	Database Object
	Organization	OrganizationRequest	An entry in the ORGANIZATION table.
	OrganizationXref	OrganizationXrefRequest	An entry in the ORGANIZATION_XREF table.
	Product	ProductRequest	An entry in the PRODUCT table
	Sensor	SensorRequest	An entry in the SENSOR table
	TagAllocation	TagAllocationRequest	An entry in the TAG_ALLOCATION table
	TagAllocationLog	TagAllocationLogRequest	An entry in the TAG_ALLOCATION_LOG table
	Unit	UnitRequest	An entry in the UNIT table. A Unit could be a case, pallet or any other entity being tracked
com.sun.autoid.epcis.business			
	Container	ContainerRequest[1]	A set of rows in the CONTAINMENT table that correspond to a hierarchy of containers.
	Transaction	TransactionRequest[2]	A set of rows in the TX_LOG table grouped by TX_ID.
com.sun.autoid.epcis.dao			
	PrimaryKey		A primary key value and table name

1 Included in package com.sun.autoid.epcis.client

2 Included in package com.sun.autoid.epcis.client

Modifying RFID Information Server Tables

You can work with the RFID Information Server `Transfer` objects in the following two ways:

- Use table `Request` objects that implement a create/retrieve/update/delete (CRUD) pattern, otherwise known as a Data Accessor Object (DAO) pattern. `Request` objects implement the basic operations on the `Transfer` objects. This is the recommended method for interacting with the `Transfer` objects.
- Use the direct APIs. Using these APIs might be necessary if you extend the tables, create new tables, or want to create `Observation` object.s. The available APIs include the following objects:
  - `FindByAttrRequest`
  - `UpdateRequest`
  - `DeleteRequest`

Using Table `Request` Objects

A table `Request` object implements the CRUD pattern for an RFID Information Server table. Not all tables support the update and delete operations. Some tables do not have a natural primary key, thus a retrieval operation might return a list of matching `Transfer` objects.

Note – `Observation` objects must be created using the `UpdateRequest` object.

CODE EXAMPLE 6-6 Creating a Product

```
EpcisConnection conn = new EpcisConnection(...)
// Create Product client
ProductRequest req = new ProductRequest(conn);
//Create Product
Product create = new Product();
create.setProductId("26");
create.setManufacturerId("1");
create.setName("Test Product");
create.setGtin("00067933861108");
create.setDescription("Product to be inserted");
create.setObjectClass("3");
```

CODE EXAMPLE 6-6 Creating a Product *(Continued)*

```
// Create Product
req.create(create);
```

CODE EXAMPLE 6-7 Retrieving a Product

```
EpcisConnection conn = new EpcisConnection(...)
// Create Product client
ProductRequest req = new ProductRequest(conn);
Product p2 = req.get("26");
if(p2 != null) {
 System.out.println("Found Product: " + p2.getDescription());
}else {
 System.out.println("Product # 26 not found!");
}
```

CODE EXAMPLE 6-8 Updating a Product

```
EpcisConnection conn = new EpcisConnection(...)
// Create Product client
ProductRequest req = new ProductRequest(conn);
Product p2 = req.get("26");
if(p2 != null) {
 p2.setDescription("modified product description");
 req.update(p2);
}else {
 System.out.println("Product # 26 not found!");
}
```

CODE EXAMPLE 6-9 Deleting a Product

```
EpcisConnection conn = new EpcisConnection(...)
// Create Product client
ProductRequest req = new ProductRequest(conn);
Product p2 = req.get("26");
if(p2 != null) {
 req.delete(p2);
}else {
 System.out.println("Product # 26 not found!");
}
```

Using the Update/Delete/Query Request Object

A request class represents an update, delete or query request to the Information Server. All request classes extend the abstract `com.sun.autoid.epcis.client.Request` class. Instances of a request class are converted to an XML format so that they can be sent over the wire.

TABLE 6-3 Request Classes and Code Examples

Class	Description	Code Examples
`UpdateRequest`	Provides methods to update the RFID Information Server database tables. Instances of this class process insert and modify operations on Unit, Product, Organization, Container, Sensor and ContainerType transfer objects.	CODE EXAMPLE 6-10 CODE EXAMPLE 6-11 CODE EXAMPLE 6-12 CODE EXAMPLE 6-13
`DeleteRequest`	Provides methods to delete entries from the RFID Information Server tables. For most tables, the class uses a PrimaryKey object to identify the entry to delete.	CODE EXAMPLE 6-14 CODE EXAMPLE 6-15 CODE EXAMPLE 6-16

CODE EXAMPLE 6-10 Inserting a Unit

```
EpcisConnection conn = new EpcisConnection(...)
UpdateRequest updateReq = new UpdateRequest(conn);

Unit unit = new Unit();
unit.setEpc("urn:epc:id:gid:2.1.1");
unit.setExpiryDate(Calendar.getInstance());
unit.setProductId("1");
unit.setUnitType("ITEM");
unit.setAttr1("192.168.1.2");    // persists the non-EPC data
UpdateResponse updateResp = updateReq.add(unit);
```

CODE EXAMPLE 6-11 Inserting a Transaction

```
ArrayList epcs = new ArrayList();
epcs.add("urn:epc:id:gid:1.402.1");
epcs.add("urn:epc:id:gid:1.301.2");
```

CODE EXAMPLE 6-11 Inserting a Transaction *(Continued)*

```
Transaction trans = new Transaction("PO-909", Calendar.getInstance(),null,
   epcs);
updateResp = updateReq.createTransaction(trans);
```

CODE EXAMPLE 6-12 Inserting Observations Using the `PML` Method

```
DeltaEvent deltaEvent = ...
Sensor sensor = EventUtil.toSensor(deltaEvent);
UpdateResponse updateResp = updateReq.postPML(sensor);
```

CODE EXAMPLE 6-13 Insert Observations Using the `ValueObject` Method

```
Observation obs = new Observation(new ObservationLogVO());
 obs.setSensorEpc("urn:epc:id:gid:1.1.1");
 obs.setObservationType("NewExternal");
 obs.setObservationValue("urn:epc:id:sgtin:0084691.142752.405");
 obs.setTimestamp(Calendar.getInstance());
 obs.setAttr1("Some test junk");

 ArrayList voList = new ArrayList();
 voList.add(obs);
 UpdateRequest updateReq = new UpdateRequest(conn);
 UpdateResponse updateResp = updateReq.add(voList);
```

CODE EXAMPLE 6-14 Deleting a Container

```
DeleteRequest deleteReq = new DeleteRequest(conn);
PrimaryKey  pk =
new PrimaryKey("urn:epc:id:gid:1.402.1", "CONTAINMENT");
ArrayList pkList =new ArrayList();
pkList.add(pk);
DeleteResponse deleteResp =
deleteReq.deleteByPk(pkList);
```

Note – Containers can be deleted by specifying the parent EPC of the container. The method only deletes the contents of the top most container. If the children of the parent EPC are containers then their contents are not deleted.

CODE EXAMPLE 6-15 Delete a Unit

```
PrimaryKey pk = new PrimaryKey("urn:epc:id:gid:1.103.1", "UNIT");
ArrayList pkList =new ArrayList();
pkList.add(pk);
DeleteResponse deleteResp = deleteReq.deleteByPk(pkList);
```

CODE EXAMPLE 6-16 Delete a Transaction

```
ArrayList txIdList = new ArrayList();
txIdList.add("PO-909");
DeleteResponse deleteResp = deleteReq.deleteTxById(txIdList);
```

Note – The TX_LOG table does not have a primary key. The transaction to delete is identified by the TX_ID value.

Querying RFID Information Server Database Tables

Query conditions are expressed as simple attribute comparisons. The valid comparators are `eq`, `lt`, and `gt`. These comparators can be used when comparing the values of fixed attributes. When comparing the value of an extended attribute, the only valid operator is `eq`. If more than one condition is specified, append each comparator by using the AND operator. If no conditions are specified, then the query returns all the entries in the selected table.

The following table lists the query classes and code examples.

TABLE 6-4 Query Request Classes and Code Examples

Class	Description	Code Example
`FindByAttrRequest`	Handles queries on a specified table in the RFID Information Server	CODE EXAMPLE 6-17 CODE EXAMPLE 6-18 CODE EXAMPLE 6-19
`ContainmentRequest`	Handles queries on the CONTAINMENT table. A containment query condition can express a parent or child relationship. It can also specify if the search should recursively return all the EPCs in the hierarchy or only the immediate EPCs.	CODE EXAMPLE 6-20 CODE EXAMPLE 6-21

CODE EXAMPLE 6-17 All Transactions in the TX_LOG Table

```
FindByAttrRequest findReq =
new FindByAttrRequest(conn, "TX_LOG");
FindByAttrResponse findResp = findReq.process();
```

CODE EXAMPLE 6-18 Query for a specific EPC from the OBSERVATION Table

```
FindByAttrRequest findReq =
new FindByAttrRequest(conn, "OBSERVATION_LOG");
findReq.addCondition
("OBSERVATION_VALUE", "urn:epc:id:gid:1.402.1", "eq");
findResp = findReq.process();
```

CODE EXAMPLE 6-19 Conditional Query From the UNIT Table

```
FindByAttrRequest findReq =
new FindByAttrRequest(conn, "UNIT");
findReq.addCondition
("EXPIRY_DATE", Calendar.getInstance(), "gt");
findReq.addCondition("PRODUCT_ID", new Integer(1), "eq");
findResp = findReq.process();
```

CODE EXAMPLE 6-20 Querying for the Parent of an EPC Recursively

```
ContainmentResponse contReq = new ContainmentRequest(conn);
ContainmentResponse contResp =
contReq.process("urn:epc:id:gid:1.103.1",
ContainmentRequest.PARENT_OF,
true);
```

CODE EXAMPLE 6-21 Querying for the Immediate Child EPCs of a Given EPC

```
ContainmentResponse contResp =
contReq.process("urn:epc:id:gid:1.402.1",
ContainmentRequest.CHILD_OF,
false);
```

Processing RFID Information Server Responses

A response message from the Information Server is marshalled into a response object. All response classes extend `com.sun.autoid.epcis.client.Response`. The following table lists the response classes with examples.

TABLE 6-5 Response Classes and Code Examples

Class	Description	Code Example
`UpdateResponse`	Represents the status of the update request. Failure status is captured as a string. The status reported by a JDBC driver is returned verbatim	CODE EXAMPLE 6-22
`DeleteResponse`	Represents the status of a delete request. Failure status is captured as a string.	CODE EXAMPLE 6-23

TABLE 6-5 Response Classes and Code Examples *(Continued)*

Class	Description	Code Example
`FindByAttrResponse`	Collects all the transfer objects that are returned as a result of a `FindByAttrRequest`. The transfer objects are generally returned as `ArrayLists`. The `ArrayList` may contain zero or more elements.	CODE EXAMPLE 6-24 CODE EXAMPLE 6-25
`ContainmentResponse`	This class can return the result as an `ArrayList` of EPCs or as a `Container` object. The former is useful if quick traversal of the result set is needed while the latter preserves the hierarchical relationships between the EPCs.	CODE EXAMPLE 6-26 CODE EXAMPLE 6-27

CODE EXAMPLE 6-22 Testing the Success of an Update Request

```
UpdateResponse updateResp = updateReq.modify(unit);
System.out.println("ModifyUnit. Success -> " +
updateResp.success() + ". Should be true");
```

CODE EXAMPLE 6-23 Testing the Success of a Delete Request

```
DeleteResponse deleteResp = deleteReq.deleteByPk(pkList);
System.out.println("Success -> " +  deleteResp.success());
```

CODE EXAMPLE 6-24 Printing the Number of Transaction Objects Returned

```
FindByAttrResponse findResp = findReq.process();
System.out.println("Results -> Got " +
findResp.getTransactions().size() );
```

CODE EXAMPLE 6-25 Printing the Attribute Value of the Returned UNIT Object

```
FindByAttrRequest  findReq = new FindByAttrRequest(conn, "UNIT");
findReq.addCondition("EPC", "urn:epc:id:gid:1.103.1", "eq");
findResp = findReq.process();
```

CODE EXAMPLE 6-25 Printing the Attribute Value of the Returned UNIT Object *(Continued)*

```
System.out.println("Results -> Got " + findResp.getUnits().size()
);
if (findResp.getUnits().size() > 0)
{
    Unit unit = (Unit)findResp.getUnits().get(0);
    System.out.println("Got unit with manufacture date : " +
unit.getManufactureDate());
}
```

CODE EXAMPLE 6-26 Using `getResultSet()`

```
ContainmentResponse contResp =
contReq.process("urn:epc:id:gid:1.103.1",
ContainmentRequest.PARENT_OF,
true);
System.out.println("Results -> Got " +
contResp.getResultSet().size() );
```

CODE EXAMPLE 6-27 Using `getRoot()`

```
ContainmentResponse contResp =
contReq.process("urn:epc:id:gid:1.103.1",
ContainmentRequest.PARENT_OF,
true);
System.out.println("Results -> Got " + contResp.getRoot().getEpc()
);
```

Handling Exceptions

Use the classes of the package, `com.sun.autoid.epcis.EPCISException`. `EPCISException` inherits and extends the Java Standard Edition standard `Exception` class.

`EPCISException` is defined as shown in the following code example:

CODE EXAMPLE 6-28 Define `EPCISException`

```
public class EPCISException extends Exception {
...
    public EPCISException(String message) {
```

CODE EXAMPLE 6-28 Define `EPCISException` *(Continued)*

```
        super (message);
    }
}
```

How to Catch an `EPCISException` Error

A program can catch the `EPCISException` error by using a combination of the `try`, `catch`, and `finally` blocks as shown in the following code example:

CODE EXAMPLE 6-29 Catch `EPCISException`

```
public class MyQuery {

    public static void main(String[] args) {
        EpcisConnection conn = null;
        String epcisUrl = null;
        String httpProxyHost = null;
        String httpProxyPort = null;
        String epcisUsername = null;
        String epcisPassword = null;

        // get the properties for EpcisConnection
        ...
        try {
             conn = new EpcisConnection(
                     epcisUrl,
                     httpProxyHost,
                     httpProxyPort,
                     epcisUserName,
                     epcisPassword);
             // querying EPCIS
           ...
         } catch (EPCISException e) {
         }
         finally {
             ...
         }
         ...
     }
}
```

How to Throw an `EPCISException` Error

A program can use `EPCISException` to indicate an error occurred. To throw an `EPCISException`, you use the `throw` statement and provide an exception message as shown in the following code example:

CODE EXAMPLE 6-30 Throw Internal Exception

```
public class FindByAttrRequest extends Request {
    ....

    /**
     * Process the query and get a response synchronously.
     * Called after the conditions have been set. This results in
     * a request to the EPCIS
     * @return The response object.
     */
   public FindByAttrResponse process() throws EPCISException{
       FindByAttrResponse findByAttrResp = null;
       EpcisMsgXML response = conn.synchAction (request);
       findByAttrResp = new FindByAttrResponse (response);
       reset();
       return findByAttrResp;
   }
   ....
}
```

CHAPTER 7

PML Utilities

This chapter describes the Sun Java System RFID Software PML Utilities. The following sections are included:

Introduction

The purpose of the core physical markup-language (PML Core) is to provide a standardized format for the exchange of data captured by the sensors in an RFID infrastructure, for example, RFID readers. This data is exchanged between the Event Manager and other applications. PML Core provides a set of XML schemas that define the interchange format for the transmission of the data values captured.

The PML Utilities Java library provides helper classes to parse and manipulate PML Core messages. This library is intended for use in any application that interfaces with the Event Manager. These utilities are located in the file `sun-rfid-common.jar`. TABLE 7-1 lists the default locations of this file for the supported platforms.

TABLE 7-1 Location of PML Utilities JAR File

Platform	Location
Solaris OS	`/opt/SUNWrfid/lib`
Linux	`/opt/sun/rfidem/lib`
Microsoft Windows	`C:\Program Files\Sun\RFID Software\rfidem\lib`

Supported version is PML_Core v1.0 (AutoID Center recommendation 15 September, 2003).

Capturing Tag Observations Using PML Core

This section describes a sample core message for capturing tag observations.

CODE EXAMPLE 7-1 Sample Core Message for Capturing Tag Observations

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- The root element of PML Core  -->
<Sensor xmlns="urn:autoid:specification:interchange:PMLCore:xml:schema:1">
<!-- The EPC of the reader specified in the reader adapter properties -->
    <ns1:ID xmlns:ns1=
"urn:autoid:specification:universal:Identifier:xml:schema:1">urn:epc:id:gid
:1.1.100
    </ns1:ID>
<!-- The root element for an observation -->
  <Observation>
     <ns2:ID xmlns:ns2=
   "urn:autoid:specification:universal:Identifier:xml:schema:1">2
     </ns2:ID>
<!-- The time when the observation was recorded by the reader
adapter -->
     <DateTime>2004-05-21T18:28:16.633-07:00</DateTime>
<!-- If the PML is generated from a Delta Event the value of Command is
either TagsIn or TagsOut -->
     <Command>TagsOut</Command>
<!-- A tag observation -->
     <Tag>
<!-- The EPC of the observed tag in identity URI format -->
      <ns3:ID xmlns:ns3=
"urn:autoid:specification:universal:Identifier:xml:schema:1">urn:epc:id:gid
:1.1.110</ns3:ID>
    </Tag>
    <Tag>
      <ns4:ID xmlns:ns4=
"urn:autoid:specification:universal:Identifier:xml:schema:1">urn:
epc:id:gid:1.1.105</ns4:ID>
    </Tag>
```

CODE EXAMPLE 7-1 Sample Core Message for Capturing Tag Observations *(Continued)*

```
    <Tag>
       <ns5:ID xmlns:ns5=
"urn:autoid:specification:universal:Identifier:xml:schema:1">urn:epc:id:gid
:1.1.104</ns5:ID>
    </Tag>
  </Observation>
</Sensor>
```

PML Utilities Packages

This section describes the following PML utilities packages:

- Package - `com.sun.autoid.pmlcore.pml`
- Package - `com.sun.autoid.pmlcore.pmlparser`

PML Core Package

The `com.sun.autoid.pmlcore.pml` package is generated from the PML Core XML Schema document using the JAXB compiler. The generated classes can be used to traverse an existing Java object graph or to create a new one. See CODE EXAMPLE 7-2.

CODE EXAMPLE 7-2 Sample XML to Create a New PML Message

```
/**
      * Create a sample PML Core XML message.
      */
    public SensorType createPMLCore() {
        SensorType sensor = null;
        try {
            PmlParser pmlParser = new PmlParser();
            ObjectFactory objFactory =
pmlParser.getPMLObjectFactory();
            sensor = objFactory.createSensor();

            /* Create the reader EPC */
            IdentifierType idType =
objFactory.createIdentifierType();
            idType.setValue("urn:epc:id:gid:1.700.1");
            sensor.setID(idType);
```

CODE EXAMPLE 7-2 Sample XML to Create a New PML Message *(Continued)*

```
            /* Create the Observation object */
            List obsList = sensor.getObservation();
            ObservationType obs =
objFactory.createObservationType();
            obsList.add(obs);

            /* Timestamp of the observation */
            obs.setDateTime(Calendar.getInstance());
            /* The command element is optional.
                if specified it is either :
                TagsIn or TagsOut */
                obs.setCommand("TagsIn");
            /* Observation ID is currently ignored */
            idType = objFactory.createIdentifierType();
            idType.setValue("1");
            obs.setID(idType);

          /* create Tags and assign them to the Observation object
*/
            TagType tag = objFactory.createTagType();
            idType = objFactory.createIdentifierType();
            idType.setValue("urn:epc:id:gid:10.10.1");
            tag.setID(idType);
            obs.getTag().add(tag);

            tag = objFactory.createTagType();
            idType = objFactory.createIdentifierType();
            idType.setValue("urn:epc:id:gid:10.10.2");
            tag.setID(idType);
            obs.getTag().add(tag);
            /* debug */
            System.out.println("Sensor BEGIN");

SensorUtil.dump((com.sun.autoid.pmlcore.pml.Sensor)sensor);
            System.out.println("Sensor END");

        } catch (JAXBException jbe) {
            jbe.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sensor;
    }
```

PML Parser Package

TABLE 7-2 Database Tables

Java Class	Descriptor
`PmlParser`	A Parser that reads Product Markup Language and creates a Java Object Graph.
`SensorUtil`	A class to dump the Sensor JAXB Tree.

Unmarshalling an XML File Using the PmlParser

CODE EXAMPLE 7-3 Sample XML to Unmarshall a PML Core XML File

```
/* Unmarshall a PML Core XML file */
    public Sensor pmlCoreFromFile() {
        Sensor sensor = null;
        try {
          /* Create an instance of PmlParser */
            PmlParser pmlParser = new PmlParser();
            /* Call the unmarshall method */
            sensor = pmlParser.unmarshalPML(new
File("./pml_sample.xml"));
            /* debug message */
            System.out.println("Sensor BEGIN");
            SensorUtil.dump(sensor);
            System.out.println("Sensor END");
        } catch (JAXBException jbe) {
            jbe.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sensor;
     }
```

Class Path Requirements

The JAR files (JAXB 1.0.4 and its dependant jar files from Java WSDP 1.5) that are used to compile and run an application that uses the PML utilities are shown in TABLE 7-3.

TABLE 7-3 PML Utilities Jar Files

`$JWSDP_HOME/jaxb/lib/jaxb-api.jar`
`$JWSDP_HOME/jaxb/lib/jaxb-impl.jar`
`$JWSDP_HOME/jaxb/lib/jaxb-libs.jar`

`$JWSDP_HOME/jwsdp-shared/lib/namespace.jar`
`$JWSDP_HOME/jwsdp-shared/lib/relaxngDatatype.jar`
`$JWSDP_HOME/jwsdp-shared/lib/jax-qname.jar`
`$JWSDP_HOME/jwsdp-shared/lib/xsdlib.jar`

`$JWSDP_HOME/jaxp/lib/jaxp-api.jar`
`$JWSDP_HOME/jaxp/lib/endorsed/dom.jar`
`$JWSDP_HOME/jaxp/lib/endorsed/sax.jar`
`$JWSDP_HOME/jaxp/lib/endorsed/xercesImpl.jar`

UML Class Diagram For PML Package