Rifidi Edge Server Developer's Guide

Version 1.0 (Rifidi Edge Server version 1.2)

September 2010

Table of Contents

Overview 4

Getting Started with the SDK 4

Overview of SDK 4

Documentation 4

Setting up a Development Environment 4

Importing a Project Template 4

Running the project 5

Setting up Configuration Files 5

Viewing and Modifying the Rifidi Source Code 5

Rifid Edge Server Architecture 6

Sensor Abstraction Layer 6

Application Engine Layer 6

Communication Layer 7

Sensor Layer 7

Creating a New Sensor Adapter 8

Anatomy of a Sensor Plugin 8

Sensor Sessions 9

AbstractSensorSession 10

AbstractIPSensorSession 10

AbstractServerSocketSensorSession 10

AbstractSerialSensorSession 11

Sensors 11

Sensor Factories 11

Persistence 11

General Purpose I/O 11

Tag Writing 12

Application Layer 12

Rifidi Application API 12

Lifecycle Management 14

Configuration Management 14

Esper Management 16

Plugging into the OSGi console 17
Rifidi Services 17
Read Zone Monitoring Service 18
Stable Set Service 18
Unique Tag Batch Interval Service 18
Unique Tag Interval Service 19
Sensor Status Monitoring Service 19
Diagnostic Applications 19
GPIO 19
Serial 20
Tags 20
Tag Generator 20
Integration Layer 21
JMS 21
Databases 22
RMI 22
Webservice 22
Exporting and Deploying 22
Example: Northwind 23
Northwind Business Rules 23
Setting up and running the Northwind Example 24
Northwind Application Architecture 25

Overview

The Rifidi Edge Server is an application platform that provides developers with a way to quickly develop and deploy RFID applications. This document describes the features and tools available to application developers in the Rifidi Edge Server Standard Development Kit (SDK).

Getting Started with the SDK

This chapter will help you get started using the Rifidi Edge Server SDK. It describes the structure of the SDK, how to set up a development environment, and how to create your first Rifidi application project.

Overview of SDK

The SDK contains the following files and folders:

- examples – Contains example Rifidi applications including a template to help you get started
- lib – Contains all the code necessary to run the Rifidi Edge Server.
- docs – Contains all the documentation
- launch file – The default run configuration to run the edge server from within eclipse
- target file – The file that tells eclipse where to find the necessary dependencies to run the edge server

Documentation

There are several places to look for specific documentation needs

- The User Documentation is a PDF that explains how to run and control the Rifidi Edge Server
- The Developer Documentation (this document) explains how to develop reader adapters and plugins for our Edge Server
- The Javadoc for the Rifidi API can be found online at http://www.rifidi.org/javadoc/edge1.2.0
- The wiki has some helpful pages to answer some common questions that users and developers have. It can be found at http://wiki.rifidi.org
- The forums provide a way for users and developers to ask questions. It can be found online at http://forums.rifidi.org

Setting up a Development Environment

In order to develop a Rifidi application, you will need to use Eclipse. Please follow the instructions on this wiki page to import the SDK into your eclipse workspace:

http://wiki.rifidi.org/index.php/Edge_Server_Development_Environment

Importing a Project Template

Now that you've set up your development environment, you will want to create a Rifidi application project. Instead of following step by step instructions on how to set up an application, it is recommended that you import a project template from the examples directory in the SDK. To do this,

1. File->Import
2. Choose General->Existing Projects into Workspace
3. Click the Browse button next to “Select root directory”
4. Browse to the SDK directory in the workspace folder
5. Select ‘org.rifidi.app.template’
6. Select ‘Copy projects into workspace’
7. Click Finish

Running the project

Once you’ve imported the template, modify it by putting a print line in the start method so that you will see that the project is running and started. Now you can run the edge server from within eclipse.

1. Open the run configuration (Run->Run Configurations)
2. Select the ‘org.rifidi.app.template’ project in the run configuration
3. Click Apply and Run

At this point you should see log output in your console to indicate that the Edge Server has been launched. At some point you should also see the debug output that you added to the template.

Setting up Configuration Files

The Rifidi Edge Server can make use of several configuration files when it starts up if they are available on the system. You can set the path to these configuration folders using the ‘org.rifidi.home’ system property. By default, it is configured to point to the Rifidi-SDK/RifidiHome directory.

- sensorconfig/rifidi.xml – a file to save reader adapter configurations
- logging.properties – a file that controls the logging output of the edge server
- applications folder – contains properties for each application that starts up. For more information see the section in this document on properties for applications
- When running the edge server from within eclipse, you do not use a edgeserver.ini file for system properties. Instead, put the system properties in the run configuration.

Notice that when the edge server starts up, it prints a line that says “All Rifidi configuration paths relative to <path>”. This is where the Rifidi edge server is looking for the properties listed above.

Viewing and Modifying the Rifidi Source Code

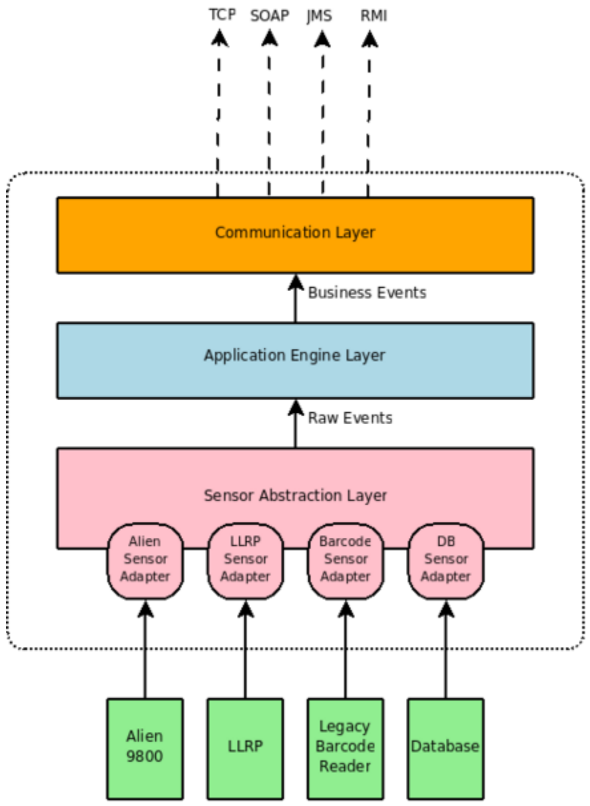
Because the Rifidi source code is included in the SDK as source plugins, you do not need to download the source code separately to view or modify it. There are two main ways to view the source code.

If your application is using a class or interface from the SDK, you can simply right click on the class in the source code and select “Open Declaration”. This will let you view the source code for the class.

If you want to see an entire source code plugin, open the plugins view in eclipse (Window->Show View->Other->Plugins). Now right-click on the plugin that you want to view and select import as->Source Project. This will allow you to not only view the source for the entire plugin, but also allow you to make changes to the source. To run the edge server with your new changes, open up the run configuration, select the plugin from the workspace and deselect it from the target platform.

Rifid Edge Server Architecture

This chapter explains the architecture of the Rifidi Edge Server at a high level. The Edge Server is broken up into three conceptual layers. The Sensor Abstraction Layer provides a common API to integrate with sensors to collect various kinds of data from them. The Application Engine Layer performs custom business processing rules on the data. The Communication layer (sometimes referred to as the integration layer) provides a means to integrate the business events collected in the Application layer with other systems (such as databases or ERP systems).



Sensor Abstraction Layer

The purpose of the edge server is to connect to any kind of sensors (e.g. RFID readers, Barcode readers, Mobile Devices) and collect information from them. In many scenarios, this consists of connecting to a Gen2 fixed reader (such as Alien 9800, Motorolla LLRP, etc), and collecting EPC information. However, the edge server is designed in a way so that so that it can collect many kinds of data (active, passive, etc) from many kinds of devices. This layer allows users to connect to devices in a sensor-agnostic way to collect the kind of data required for the application.

Application Engine Layer

For most applications it is not desirable to save every event that the sensors produce. Many sensors can send 1,000 of events a second, a large number of which might be duplicates. Most applications are interested in events that are one-level higher than the raw events produced by sensors. For examples, an ERP system is probably interested in the event of a box arriving in area 1, and it is not desirable for the ERP system to do the work of filtering and processing all of duplicate reads the sensor produces.

Complex Event Processing (CEP) is a paradigm of viewing data as ephemeral events (i.e. a stream consisting of non-persisted events) and identifying meaningful (i.e. business) events from the stream using rules. Rifidi Edge Server uses a Complex Event Processor called Esper. It allows you to write queries using an SQL-like syntax. An example query to get tags from a particular reader might look something like this:

```
select * from ReadCycle where ReaderID='gate_1'
```

The application layer lets developers write custom business logic that uses Esper to filter, aggregate and process events produced by the Sensor. The applications in this layer can perform custom business logic based on the tags that are seen. For example, one application might alert a warehouse manager via an email if a tag that matches a certain pattern is seen in a particular area. Another application might correlate barcode reads with RFID tags and write the association to a database.

Communication Layer

After data has been processed, it probably needs to be handed up to some kind of application-dependent system. For example, some users might want the data to be stored in a database, others might want it to be pushed into an ERP system like SAP or handed to a Rich User Interface of some sort. The edge server has several built in connectors to use, namely JMS and Web Services (via Spring's remoting framework). However, as this is application dependant, it is possible to write your own connector (such as a TCP/IP socket connection) if the application needs it.

Sensor Layer

The sensor layer allows the edge server to connect to various kinds of sensors and collect data from them. The kinds of sensors can be wide ranging, from network-enabled Gen2 RFID readers to barcode scanners to JMS queues of events. In addition, the events can be of various types. The most common events are Gen2 RFID, but other event types can be collected as well, such as barcodes and GPIO events.

In addition to collecting events, the sensor layer allows some level of sensor configuration. The level of configuration depends on the capabilities of the sensor and how much works has been put into the adapter. For the most part, however, sensor configuration is normally handled by a technician when first setting up the sensor. The sensor is either then configured to send events back to the edge server or the edge server will poll the sensor for events. For the vast majority of the cases, once a sensor had been configured, it will not change much. Thus, it is easiest in most cases to configure the sensor using a tool provided by the sensor’s manufacturer.

This section describes the main components of a sensor adapter for the Rifidi Edge Server. It does not give a step-by-step walkthrough on how to build an adapter, since the easiest way to learn this is by looking at the source code for adapters that ship with the Edge Server. In addition the sensor API’s source code is well documented, so it is recommended to look into that. This chapter serves as a guide on how the various pieces of the sensor API work together.

Creating a New Sensor Adapter

The Rifidi Edge Server ships with adapters for several popular RFID readers out of the box. If you need to create a new sensor adapter, there are a couple of things to consider:

- What is the communication channel that the sensor uses? If it is TCP/IP or serial, the sensor API has classes that you can extend. Many sensors fall into this category.
- Does the manufacturer provide a java API? Some sensor manufactures provide a java library that will parse and encode messages. If they do not, you will need to write this code yourself
- Will you need to poll the sensor for data, or will the sensor be configured to automatically send back data?
- Will you need GPIO support?
- Will you need tag writing support?
- What kind of data will the sensor send back? The Edge Server has support for Gen2 tags and some barcodes, but if you are working with a special kind of data, you may need to develop some classes to represent that data.

Anatomy of a Sensor Plugin

All sensors consist of three main classes:

- A SensorSession class that creates a connection to the sensor and collects data from it. There is typically one SensorSession per Sensor object. Once a SensorSession is created, it cannot be changed.
- A Sensor class that creates and maintains SensorSessions. There is typically one Sensor object for every physical sensor. For example, if you have two Alien readers, you will create two Sensors. A sensor exposes connection properties (such as an IP and Port) to use when it creates a new session. A sensor also has an ID to identify it from the OSGi command line. For example, if you have two Alien readers, you might have two Reader IDs called Alien_1 and Alien_2.
- A SensorFactory class that creates Sensors. There is one SensorFactory per Sensor type. For example, there is one SensorFactory for an Alien plugin. If you have two Alien readers in your infrastructure, you will use the AlienSensorFactory to create both AlienSensors. The SensorFactory. A SensorFactory has an ID to identify it from the OSGi command line.

In addition, all but the most basic of sensors will have Commands. Commands interact with a sensor. For example, a poll command might ask a sensor for the tags that it can currently see. There are two types of commands. Single-Shot (or one-time) commands are intended to be executed only once. Repeated commands are intended to be scheduled for repeated execution. There are three classes every command will need

- Command – The command class is a runnable which is executed by a session. It is intended that the command should execute quickly. That is, it should avoid sleeping or long running loops. Once a command is created, it cannot be changed.
- CommandConfiguration – The CommandConfiguration creates Commands. Like the Sensor, it can expose properties which it uses when creating the Command. CommandConfigurations have IDs used for controlling them from the OSGi command line.
- CommandConfigurationFactory – A factory that produces CommandConfigurations. There is one factory per CommandConfiguration type.

Finally, sensors that need to allow applications to access their GPIO capabilities can implement the GPIOService. This service allows application-level access to querying GPI state and setting GPO state.

Sensor Sessions

The Sensor Session is the most important part of a sensor adapter. It has three main roles:

1. Connection Logic – The sensor session contains the logic for connecting to and maintaining a connection with a sensor. Well written sensors sessions should normally detect when a sensor has been disconnected and attempt to reconnect if possible.
2. Command Execution – The sensor sessions have to ensure that commands issued to sensors are carried out in a thread-safe way.
3. Protocol Parsing – It is the responsibility of the sensor session to parse incoming messages according to the protocol that the sensor uses.

The Sensor API in the Rifidi Edge Server provides several base classes that implementations can extend which handle common cases. For example, because TCP/IP is a typical protocol used for connecting to readers, the API supplies an abstract class that handles TCP/IP connections robustly (it can detect if the socket is closed and attempts to reconnect). In addition, several other classes are provided that handle various kinds of connections.

Some kinds of command messages are often necessary to control the sensor. For example, some sensors require a command to tell sensor to start sending back tag reads. For others, it is necessary to poll the sensor to ask if it has seen any new data. It is the job of the sensor session to ensure that only one command is issued at a time and to allow commands to be scheduled for repeated execution if necessary.

When developing a sensor adapter, most of the work will typically go in to protocol parsing, since this is what varies widely from sensor to sensor. For example, the LLRP sensor sends back byte messages that are encoded according the LLRP specification. The Alien reader sends back clear-text strings. Most barcode readers will send back the barcode that they read.

The end goal of a sensor session is to parse events that comes back from a sensor and put them into the Esper event engine. From there, the Application Event Layer can process the events. Since many applications built on top of the Rifidi Edge Server use Gen2 RFID readers, the Rifidi API provides a common structure for Gen2 tag data. For more information on this, please see http://wiki.rifidi.org/index.php/ReadCycle_Class_Hierarchy

One last note on sensor session development: It is common for network-enabled readers to have one channel that is intended for interactive communication in a request-response mode, and one or more passive channels which the reader uses to send events on. For example, the Alien reader has an interactive channel on port 23. You can connect to this port and send commands and receive responses. You can then configure the reader to send back tag data and GPIO data to different ports. The Alien reader adapter in the edge server has one master sensor session (the interactive session) and has two "slave" sessions (one for passively receiving tag data, and one for passively receiving GPIO data). This master-slave pattern is common and typically works out well.

The following sections describe abstract classes that can be used when developing a Sensor Session class.

AbstractSensorSession

The AbstractSensorSession contains logic for executing commands. It uses A ScheduledThreadPoolExecutor to schedule commands. This logic should work for every SensorSession. It does not contain any logic for connecting to readers.

Many sensor adapters will be able to make use of a subclass of AbstractSensorSession that already handles connection logic. For example, readers that connect via TCP/IP or Serial can most likely inhert from a subclass. You should only have to subclass AbstractSensorSession directly if you need your sensor communicates via some other kind of protocol. In addition, some sensor manufactures will provide a java jar which contains and API for connecting to their reader. If this API contains connection logic, you can inherit from AbstractSensorSession directly and rely on the manufacturer's API to do connection logic.

AbstractIPSensorSession

This class handles the connection logic for a sensor which communicates via TCP/IP. It is used for interactive sessions (it can both send and receive messages). However, you should most likely not subclass this class directly. Instead, subclass either the AbstractPollIPSensorSession or the AbstractPubSubIPSensorSession class. The difference between these two classes is the way in which messages from the sensor are delivered. If messages from the reader are delivered synchronously (that is that you expect a response for every command that you send), then use the Poll session. If messages are delivered asynchronously (that is for each command you send to the reader, the reader might send back multiple responses or none at all), use the PubSub session.

AbstractServerSocketSensorSession

Use this class for passive TCP/IP sessions. It will open up a socket and passively listen for a sensor to send data to it. It cannot reply back to the sensor. It is often uses for "slave" sessions such as the GPIO session for an Alien reader. It can also be used to integrate hand held readers into the edge server. A handheld reader that is connected to a wifi network could just send data to the port for example.

AbstractSerialSensorSession

This sensor session uses the RXTX library to connect to a serial port. With a serial connection it is not possible to detect if a connection has been broken or not.

Sensors

The purpose of the Sensor class is to create and maintain an immutable instance of a sensor session. Typically, a sensor will have only one session. Once the session is created, it should not be changed. If something needs to change on the session (such as the port that the session is connected to), the session must be destroyed, the property must be changed on the sensor, and the session must be created again.

Sensors contain connection properties (such as hostname, port, etc). These properties are exposed to the OSGi command line via getter and setter methods. For example, a sensor might have a method with this signature

```
Public void setPort(Integer port)
```

This sensor exposes the port property to the OSGi command line. A user can then change the port with the 'setproperties' OSGi console command.

Each sensor has an ID that is used to identify it on the OSGi command line. If you type in 'readers' you will get a list of the sensors that have been created.

There is normally one Sensor instance per physical sensor device. Subclasses should extend AbstractSensor.

Sensor Factories

A Sensor Factory creates instances of Sensors. There is one instance of the SensorFactory per sensor adapter type. For example, there is one AlienSensorFactory and one LLRPSensorFactory available when the edge server starts up. These factories can create multiple instances of Sensors.

Each sensor factory has an ID that is used when creating new Sensors.

Subclasses should extend AbstractSensorFactory.

Persistence

Sensors and Sessions are persisted to an XML file when the 'save' OSGi command is issued. When the edge server restarts these sensors and session will be recreated, and the session will resume their saved state.

General Purpose I/O

Many popular RFID readers offer General Purpose I/O capabilities. This allows the readers to interact with other devices. For example, a reader might be configured to start reading when a photo eye detects a forklift in the near vicinity. Using a photo eye as input is an example of GPI. Other times, a reader might want to send some output to another device based on some logic. For example, a reader might want to light up a green light if a box belongs in a certain area and a red light if the box does not belong there. This is an example of GPO.

Often, GPIO data needs to be used in the application layer. For example, suppose different business logic should be executed depending on which photo eye saw a forklift. In this case, the reader should be configured to send back GPI events, and the sensor session can put the GPI events into the Esper engine. Then, the application can execute the proper logic based on the events that it saw in Esper.

GPO differs from GPI in that it necessitates that the application actively controls the reader rather than passively listen for events from the reader. In the previously mentioned green light/red light example, only the application knows if a box belongs in a certain area or not (presumably from a database look up). The sensor adapter and physical reader are just reporting back tags. This means the application must tell the sensor to turn on a green light or a red light.

To meet this need a sensor adapter can implement the AbstractGPIOService, which has a few abstract methods in it that allows control over the GPIO capabilities of the reader. The sensor can then put this service into the OSGi registry, and the application can look it up and use it as it needs to. For more information about how to implement this, please see the AlienGPIOService or AwidGPIOService classes.

Tag Writing

Tag writing is not currently supported by any reader. Tag writing, like turning on a GPO light, needs to happen in application logic. Thus an interface could be developed (similar to the AbstractGPIOService) which would allow an application to control the reader to write tags.

Application Layer

The purpose of the Sensor Layer is to collect data from sensors and put them into Esper. The purpose of the application layer is to perform business logic on the data that the sensors collect. The application layer is intended to be general enough to support a wide variety of applications, but provide some tools that are common to many applications.

This section starts out by taking an in-depth look at the Rifidi Application API and how to use it. It then details the Rifid Services, which capture common patterns seen across many RFID applications. These services allow you to create applications without have to write Esper statements. Finally it describes some of the application that ship with the Edge Server which are useful for testing and debugging applications and sensor plugins.

Rifidi Application API

Rifid Applications are at the heart of the Application Layer; these classes are used to add custom Esper statements to look for events, subscribe to Rifidi services, and integrate with existing infrastructure such as databases and JMS queues.

In order to provide consistent development, deployment, and management of Rifidi Applications, all applications should extend **`AbstractRifidiApp`**. This class provides a base set of services to Rifidi Applications, including:

- Life cycle management (starting and stopping the application)
- Configuration management (Using property files to provide input parameters to the application)
- Esper management (Ensuring that Esper is used correctly)
- Plugging into the OSGi console (Allowing your application to be controlled by the OSGi command line)

There are two pieces to every application. The first is the Application class itself.

```
Public class MyApp extends AbstractRifidiApp{

   Public MyApp(){
         super(“group”, “app”);
   }

   @Override
   public void _start(){
         //insert code here to create esper statements or subscribe
        // to rifidi services
   }

   @Override
   public void _stop(){
         //insert any clean up code here
   }
}
```

The second part is the Spring XML which creates the application, injects the application with any dependencies it requires (such as rifidi services or database connections, etc) and registers the application in the OSGi service registry. This xml file goes in the “META-INF/spring” folder in the bundle. The required XML namespaces have been removed from the following example for the sake of brevity.

```
<!-- Create the application object -->
<bean id=”app1” class=”com.mycompany.MyApp”/>

<!-- register the app in the OSGi service registry -->
<osgi:service ref=”app1”
interface=”org.rifidi.edge.core.app.api.RifidiApp”/>
```

The best way to get started with your own application is to import the Template application from the SDK and begin modifying it. In addition, there are several well-documented example applications in the SDK which demonstrate many of the features of the Application API. Exploring those examples are the best way to get a feel for how to code using the API.

You can start, stop, and monitor applications using the **`RifidiAppManager`**. Open up the OSGi console and type

```
>apps
```

You will see a list of applications printed out. Each line contains

- The application ID (a numeric ID)
- The application group
- The application name
- The state of the application (started or stopped)

You can start a stopped application using the **`startapp`** command. Likewise, you can stop a started application using the **`stopapp`** command.

The rest of this section goes into more depth about the services offered by the **`AbstractRifidiApp`** abstract class.

Lifecycle Management

Applications can be in one of two states: started or stopped. The **`AbstractRifidiApp`** class provides two methods for developers to override. The **`_start()`** method is where most of the application code belongs. It is used to add Esper statements and listeners, add custom Esper event types, subscribe to Rifidi Services, or do any other work that needs to be done when starting the application. The **`_stop()`** method is used to do any cleanup work necessary, such as unsubscribing from Rifidi Services.

When a bundle that contains one or more applications starts up, it should register its application(s) with the AppManager. It does this by exporting the application into the OSGi registry under the RifidiApp interface in the spring XML. The AppManager automatically keeps track of any object in the OSGi registry that is exported under the **`RifidiApp`** interface. It assigns this application and ID and automatically starts it if **`lazyStart()`** returns false. The AppManager allows users to start and stop the applications.

Configuration Management

One common need of many applications is to be able to externalize configuration properties to files so that they can be changed easily. The Rifidi Application API gives you an easy way to do this using a standard directory structure and file-naming convention.

At this point it is important to note that each Rifidi App must provide in its constructor a ‘group name’ and an ‘app name’. Groups are logical sets of applications. For example, the Acme Corporation might have one group of applications called ‘receiving’ which handle in-bound packages. The might have another group of applications called ‘exporting’ which handle out-bound packages. The main reason for using a group is that applications within a group can share configuration files.

There are three main types of configuration files that the Application API makes available to Rifidi Apps

- Property Files follow the conventional name=value format. They are read in when the application starts and are made available in the `initialize()` method.
- Read zone files are stored in the readzones directory in a group directory. Each one describes a different logical readzone. These are useful with Rifidi Services to determine which logical read zones a particular application is interested in.
- Data files are stored in the data directory. These files can contain any data the application needs. Applications can both read and write these files.

Properties

If you open up the 'applications' directory in the 'RifidiHome' directory of the SDK, you will notice several sub directories. The directories at this level are so-called group directories. The names of these directories correspond to the group names that applications provide in their constructor. If you open up a group directory, you will notice some property files. Each group directory can contain one property file whose name is the 'group name' string that the applications use in their constructor. Properties in this file will be shared with all applications in the group. In addition, there can be one property file per application. Each of these property files will share their name with the 'app name' of their corresponding application. If a property is both in the group property file and the application property file, the value in the application property file will be used.

For example, the Acme Corporation might use the following directory structure for their receiving and exporting applications. Folders are in **bold** and property files are in *italics*.

- **RifidiHome**
  - **applications**
    - **Receiving**
      - *Receiving.properties*
      - *App1.properties*
      - *App2.properties*
    - **Exporting**
      - *Exporting.properties*
      - *App3.properties*

In this example, the Acme Corporation has two groups (Receiving and Exporting) and three applications (App1, App2, and App3). App1 and App2 are both in the Receiving group. They share properties that are in the Receiving.properties file. App3 is in the Exporting group.

Applications can should access properties by calling the `getProperty()` method . This method should be called in the `initialize()` method of the Rifidi App.

All Rifid applications use a property called `LazyStart` to determine if the application should be started as soon as it is loaded. If this property is set to true, the application will not be started automatically. If it is set to false, then it will be started automatically. If the `LazyStart` variable is not defined in a property file, it defaults to false.

Read Zones

One common requirement for RFID applications is to define logical read zones for applications. For example, a particular application might only be interested in tags from antennas 1 and 2 of a certain reader. To this end, you can create readzone property files in the readzones directory of a group. These files will be read in and ReadZone objects will be automatically created.

The readzone files contain the following properties:

- readerID – the internal ID used to identify this reader. This ID corresponds to the Sensor ID used in the Sensor layer
- antennas – an optional comma-delimitated list of integers which correspond to the antennas which should be used.
- tagPattern – an optional regular expression defining a pattern which can be used to filter tags based on their IDs
- matchPattern – an optional boolean used with the tagPattern. If set to true accept only the tags that match the pattern. If set to false, filter out tags that match the pattern

The names of the readzone files follow a naming convention: **`readzone-[ID].properties`**. They start with the word 'readzone'. The word in between the '-' and the '.' characters is the ID of the readzone.

Readzones are read in when the application starts. They can be accessed using the **`getReadZones()`** method in the AbstractRifidiApp class, which returns a HashMap where the keys are the readzone IDs and the values are ReadZone objects.

For convenience, most of the Rifidi Services use ReadZone objects to define their data windows. In addition, you can use the **`buildInsertStatement()`** in the **`EsperUtil`** class to build an esper statement that will insert tags into a given window that match some supplied readzones.

Data Files

Many times, it is useful for applications to read and write files. The App API provides a mechanism to make it easy for applications to do this. Data files reside in the 'data' directory, and they share a naming convention that is similar to the **`readzones: [prefix]-[id].[suffix]`** They can be accessed using the getDataFiles() method . This method takes in a String which is the prefix. When this method is called all data files with the given prefix are read in, turned into byte arrays and made available as a hashmap. A particular file is located from within that hashmap using its ID.

Files can also be written to the data directory using the **`writeData()`** method. This method will write a new file when it is called.

Esper Management

Esper is a complex event processing engine. It allows users to insert events in the form of java objects into the engine and look for patterns in this data using queries. It is a powerful tool for building RFID applications because it allows Sensors to push events into the engine and applications to state the patterns they are looking for. Then applications are notified when an event happens that match the pattern. Because events are filtered in memory, it allows RFID applications not to rely on databases and enables an event-driven architecture.

The Application API helps developers keep up with two aspects of esper: statements and custom event types. Please consult the Esper documentation online for information about esper syntax and semantics.

Statements

Esper's query syntax is much like SQL. Applications can add these Esper 'statements' to the Esper runtime. In addition, they can define listeners to certain statements. Because it is important to keep up with which statements have been added and to make sure statements are removed when the application starts up, the Abstract Rifidi App provides two methods that should be used when adding statements and listeners.

- **`addStatement(String)`** is used to add a single statement to the esper runtime
- **`addStatement(String,StatementAwareUpdateListener)`** is used to add a statement and a listener to that statement. The updateListener will allow you to handle any events which trigger the statement.

By using these methods to add statements, there is no need to remove the statements when your application stops. This is handled automatically for you.

Custom Event Types

The Esper runtime must know ahead of time what kind of events will be put into it. There are two types of events: Java objects and Map events. Please see the Esper documentation about how to access properties from these events in statements.

The Rifidi Edge Server has several kinds of events already defined in the Esper runtime that every application can make use of, such as ReadCycle, TagReadEvent, GPIEvent, and GPOEvent. If your application needs to add its own custom event type, it can do so using any of the **`addEventType()`** methods. Using these methods will ensure that the Event Types are properly removed from the runtime when your application stops.

Plugging into the OSGi console

The OSGi console allows users to access their application via a command line. It is often useful for administration and testing purposes. Each application can contribute to the console by overriding the **`getCommandProvider()`** method. Contributing an object that implements the **`CommandProvider`** interface will allow the application to expose its own commands on the OSGi command line.

Rifidi Services

Esper is a powerful tool for building RFID applications. However, many RFID applications have common needs. For these scenarios, the Rifidi Edge Server offers 'Rifidi Services' which capture a few common

RFID use cases and allow your application to subscribe to these patterns without the need to write custom Esper statements.

In general, to subscribe to a service you need to follow these steps:

1. In the spring xml for the application, get a hold of the service that you want to subscribe to. Inject it into your application
2. In your application, implement the appropriate subscriber interface
3. In the _start() for your application, subscribe to the service passing in the subscriber and any parameters needed.
4. In the _stop() method for your application, unsubscribe from the service.

Read Zone Monitoring Service

The Read Zone Monitoring service notifies subscribers when a tag enters a particular readzone and when it leaves a read zone.

To subscribe to this service, put the following in your spring.xml file:

```
<osgi:reference id=”ReadZoneService”
interface=”org.rifidi.edge.core.app.api.service.tagmonitor.ReadZoneMonitoringService”/>
```

You should implement the ReadZoneSubscriber interface, which gives you two methods:

```
tagArrived(TagReadEvent tag)
tagDeparted(TagReadEvent tag)
```

Stable Set Service

The Stable Set Service notifies subscribers when a given amount of time has passed without any new tags having arrived. It then passes all the tags seen in that interval to the listener. This pattern is often useful when there is a concept of children tags and a parent tag. For example, imagine that you want to group items to a container. You put all the tags in the field of view of an antenna, and you want to be able to process all the tags seen as a group.

To subscribe to this service, put the following in your spring.xml file

```
<osgi:reference id=”StableSetService”
interface=”org.rifidi.edge.core.app.api.service.tagmonitor.StableSetService”/>
```

You should implement the StableSetSubscriber interface, which gives you the following method:

```
stableSetReached(Set<TagReadEvent> stableSet)
```

Unique Tag Batch Interval Service

The Unique Tag Batch Interval Service periodically notifies subscribers of unique tags that have been seen in the read zone since the last notification.

To subscribe to this service, put the following in your spring.xml file

```
<osgi:reference id=”UniqueTagBatchIntervalService”
interface=”org.rifidi.edge.core.app.api.service.tagmonitor.UniqueTagBatchIntervalService”/>
```

You should implement the UniqueTagBatchIntervalSubscriber interface, which gives you the following method:

```
tagBatchSeen(Set<TagReadEvent> tags)
```

Unique Tag Interval Service

The Unique Tag Interval Service notifies you the first time a unique tags is seen at a read zone and periodically after that if the tag is still in the read zone.

To subscribe to this service, put the following in your spring.xml file

```
<osgi:reference id=”UniqueTagIntervalService”
interface=”org.rifidi.edge.core.app.api.service.tagmonitor.UniqueTagIntervalService”/>
```

You should implement the UniqueTagIntervalSubscriber interface, which gives you the following method:

```
tagSeen(TagReadEvent tags)
```

Sensor Status Monitoring Service

This service is slightly different from the others mentioned so far. Instead of notifying subscribers about Tag events that happen, it notifies subscribers about changes to the sensor layer. It’s useful for giving your application access to when Sensor connect or disconnect.

To subscribe to this service, put the following in your spring.xml file

```
<osgi:reference id=”SensorSubscriberService”
interface=”org.rifidi.edge.core.app.api.service.sensormonitor.SensorStatusMonitoringService”/>
```

You should implement the SensorStatusSubcriber interface, which gives you the following method:

```
handleSensorStatusEvent(SensorStatusEvent event)
```

Diagnostic Applications

The Rifidi Edge Server platform ships with several applications designed to help gather information some diagnostic information and to help developers and administrators troubleshoot common problems.

GPIO

The GPIO application allows users to interact with GPIO devices (as long as the reader and Rifidi Edge Server reader adapter support GPIO). See the help menu on the OSGi console for correct usage of these commands.

- testGPI - Returns the current GPI state on a reader
- setGPO – Sets a GPO state on a reader
- flashGPO – Changes a GPO state on a reader for a certain amount of time

In addition, there are a few commands which "simulate" GPI events by inserting fake GPI events into the esper runtime. This is useful for testing and troubleshooting applications which rely on GPI events from a reader to operate without having to actually have the hardware hooked up.

- simGPIHigh – sends a GPI High event into Esper
- simGPILow – sends a GPI Low event into Esper
- simGPIFlashHigh – sends a GPI high event and then a GPI low event into Esper
- simGPIFlashLow – sends a GPI Low event and then a GPI High event into esper

Serial
The serial application exposes several commands that help you determine which serial ports are available on your machine. This is often useful when configuring a reader adapter which uses a serial protocol.

- connectSerial – connect to a certain serial port
- disconnectSerial – disconnect from a certain serial port
- listSerial – List all serial ports available on your machine

Tags
The tags application allows developers and to gather some information about the tags that the edge server is gathering.

- currenttags – lists the tags that the edge server can currently see
- recenttags - lists the tags that the edge server has seen recently
- tagrate - lists the number of tags per second that the edge server is current seeing

Tag Generator
This utility allows developers to send 'fake' tag events into Esper. It is useful when developing and testing applications so that you don't have to have real hardware hooked up to your system to send tags events into the reader.

It works using a set of two property files, both located in the RifidiHome/applications/Diagnostic/data directory.

- exposure-<ID>.properties – this file controls how the tags will be exposed to Esper. For example, it controls the rate at which tags are exposed and triggers to stop exposing them
- tags-<ID>.txt – This file contains the tags to expose along with which reader and antenna

In order to initiate the fake tag reads, use this command:

```
> startTagRunner <tagFileID> <exposureFileID>
```

Exposures Files

There are several properties in an exposure file that control how tags are exposed

- **`exposureType`**: There are two basic types of exposures: ‘rate’ and ‘delay’. A ‘rate’ type means that the exposure will attempt to put tags into esper according to a given rate. A ‘delay’ means that the exposure will expose a defined number of tags with a given delay in between each cycle.
- **`stop.count:`** A count-based stop trigger. The exposure will stop running after the defined number of tags have been exposed.
- **`stop.time:`** A time-based stop trigger. The exposure will stop running after the defined number of milliseconds
- **`random:`** If true, the exposure will pick tags from the tag file at random
- **`tagRate:`** Used with ‘rate’ exposureType. Indicates how many tags per second we should expose to esper
- **`groupSize:`** Used with ‘delay’ exposureType. Indicates how many tags should be exposed at once.
- **`delay:`** Used with ‘delay’ exposureType. Indicates number of milliseconds to pause in between exposing a group.

Tag Files

Tag files simply contain the data to expose. Each tag is on a new line. Each line has three comma-separated fields: the tagID, the readerID, and the antenna ID.

Integration Layer

Once applications have identified the business events they are interested in, they will want to integrate with existing systems. The Rifidi Edge Server provides several methods of integration out of the box. The integration layer is heavily spring based, and in general, you can use many spring technologies inside of the Rifidi Edge Server. The following section lists a few of many possible integration technologies.

JMS

The Rifidi Edge Server runs an embedded ActiveMQ broker. You can change the ActiveMQ’s settings in by editing the config/rifidi-amq.xml file.

In addition, there is already a preconfigured spring JMSTemplate that you can use to send out messages. You can get a hold of it in spring with the following code:

```
<osgi:reference id=”externalJMSTemplate”
interface=”org.springframework.jms.core.JmsTempalte” bean-
name=”externalJMSTemplate”/>
```

Databases

Out of the box, the Rifidi Edge Server ships with Derby – an embeddable database . The database starts up when the edge server starts. One way to access it is using Spring's JDBCTemplate. Please see spring's documentation on how to use the JDBCTempalte. The connection properties you need are

```
driverClassName=org.apache.derby.jdbc.EmbeddedDriver
url=jdbc:derby:DB_NAME;create=true
```

In addition, you can extend Rifidi's AbstractDBDAO class which might provide a useful interface for some use cases.

RMI

You can use Spring's RMIServiceExporter to expose an interface over RMI. Please see Spring's documentation for details.

Webservice

If you have a service that you want to expose as a WebService, you can useRifidi's JaxWsServiceExporter. The following shows how to use the service in spring. Please note that the 'service' property refers to a bean that has the @WebService annotation. Please refer to Spring's JaxWsServiceExporter for details on how to use this.

```
   <bean id="LRWebServiceDeployer"
class="org.rifidi.edge.core.utilities.JaxWsServiceExporter"
         init-method="start" destroy-method="stop">
         <property name="port" value="8080/>
         <property name="host" value="http://localhost" />
         <property name="deploy" value="true" />
         <property name="service" ref="beanID" />
   </bean>
```

Exporting and Deploying

Once your application is built and tested, you will want to export it so that it can run on an instance of the edge server. In order deploy the application there are a few steps you need to follow:

1. Export the projects as OSGi bundles out of eclipse
2. Run the bindex utility on them in order to generate a repository.xml file that is used when dynamically loading your bundle.
3. Place the jar and xml file into the application's group folder.
4. If you want the application to start immediately, add the name of the group folder to the default.ini file in the applications directory.

This is an example of the directory structure for an application whose group is 'Acme' and whose application name is 'Shipping'

- **RifidiHome**

- **applications**
    - **Acme**
        - **plugins**
            - *acme.jar*
        - *Acme.properties*
        - *Shipping.properties*
        - *repository.xml*

You can find more information about this topic on our wiki

http://wiki.rifidi.org/index.php/How_to_export_your_custom_Rifidi_application

Example: Northwind

Congratulations! You are the proud new founder of Northwind Shipping Inc. -- "delivering packages faster than a caffeinated lightning bug"™. One of your core business strategies is to out perform your competitor -- Pony Express Shipping Inc. -- by capitalizing on increased efficiencies gained by your innovative use of technology. You have heard all the hype about RFID and want to employ in it your new, state-of-the-art distribution center. You have decided to use the Rifidi Edge Server to run the RFID applications you will need in your distribution center.

This tutorial will show you how to set up an application and use all the various features provided by the Rifidi Edge application API.

Northwind Business Rules

The Northwind distribution center's current process looks like this:

1. Boxes (fitted with RFID tags) are loaded onto a pallet, which are transported via forklift (also outfitted with RFID tags) to a dock door. From the dock door, an inventory is taken.
2. The forklift will then take the packages to a weigh station, where the packages would be weighed.
3. After the weigh station, the packages would be sent out for shipping. RFID can add value to this workflow by monitoring whenever a tag enters or leaves an area (either the dock door or the weigh station), and do extra processing based on what happens when it enters and leaves an area.

A message should be printed when...

1. a tag enters or leaves an area.
2. a tag moves from the dock door to the weigh station.
3. tags are seen with their forklift tag.

Occasionally, something can go wrong and an item will do something unexpected. Error messages should be printed if...

1. Tags move from the weigh station to the dock door, stating that the tag moved backwards.
2. Tags are seen on the weigh station which was not seen at the dock door.
3. Item tags are seen without an accompanying forklift tag.

More documentation on how this was accomplished is given in the Esper documentation, as well as the

Setting up and running the Northwind Example

1. Set up the eclipse workspace and import the SDK.
2. Import the plugin from the “examples” directory in the SDK plugin.
3. Go to Run > Run Configurations and click on “Edge Server” under “OSGi Framework” on the left. 4. Select the package “org.rifidi.edge.northwind” in the “bundles” tab. Press “Apply”.
4. Now either close and run the “Edge Server” launch framework or just press “run”.
5. Type in “apps” and make sure the Northwind Application has started.

If you don't see the Northwind app at all, make sure the plugin is set to Auto-Start in the run configuration. If you see it but it is stopped, you will have to manually start it. Check the Applications section of the documentation for more help debugging problems starting applications.

After you know the application is working, stop the edge server and open the org.rifidi.edge.northwind plugin. Copy the “Northwind” folder and paste it into the “Rifidi-SDK/RifidiHome/applications” folder. Then open the folder, and open the “Northwind.properties” file. This shows all of the properties that can be adjusted for this application (see the application documentation for more on the properties files). You might need to adjust the names of the readers for the dock door and weigh station, depending on what the names of the readers are in Rifidi Edge. A more thorough explanation of each of the properties is in the file itself. The properties files for both the dock door and weigh station ReadZones are also in this folder.

After the properties files are set up correctly, run the program and create the readers you want to use. If you want, you can use the included Prototyper file with an already set up dock door reader (127.0.0.1:20000), and an already set up weigh station reader (127.0.0.1:30000). If you like, you can use any physical readers you have access to, or you can use Emulator to create the readers of your choice and then use edge to connect to them. After you have connected to the readers set in the Northwind properties file, you can begin adding and removing tags from the field of view of the readers. Here is a sample of what you can do:

1. Create 1 GID tag and 2 SSCC tags (this assumes that the “ForkliftPrefix” property is set to the default “35”).
2. As a group, move all 3 tags into the field of view of the Dock Door reader. You should see 3 “tag arrived” messages for the dock door, plus a message telling you that a forklift was seen with the other tags.
3. Delete all the tags from the field of view. You should get 3 “tag departed” messages for the dock door.
4. Move the same 3 tags onto the weigh station reader. You should see 3 “tag arrived” events for the weigh station, as well as 3 events telling you that the tags have moved from the dock door

to the weigh station. You should also get output telling you that a forklift was seen. This is only one example of what you can do; only some of the rules were used for it. Invoking the rules given above on your own, as well as studying the Esper used to create these rules will help you greatly when you are writing your own application.

Northwind Application Architecture

The Northwind application contains three parts:

1. The application class: NorthwindApp.java is where the program is set up, and all of the logic is implemented. Setting up the ReadZones, Subscribers, properties, and Esper all happens in this class. Check out the file itself for a more detailed explanation of how the application works.
2. The event classes: These are simply shell classes that are created to be passed into Esper. They contain no information except for the ID of the tag that they represent, but any events you make can contain any information.
3. The subscriber classes: These classes represent two different kinds of subscriber: one is a regular ReadZoneSubscriber, which will subscribe to ReadZones that we create and fire events based on certain criteria. The other is a StableSetSubscriber that will look for forklift tags and print output depending if a forklift is present when tags are read. For more information on how these classes work, check their respective files.

The Northwind files themselves are well-commented, and should help you get a better understanding of how applications are built and run. Other application plugins, such as the diagnostic tools (org.rifidi.edge.app.diag) should give you more examples of what you can do with applications, should you need more examples. Finally, if you need any more help with your application, go to our forums:

http://forums.rifidi.org/viewforum.php?f=35