Rifidi® Edge Server Developer's Guide

Version 3.4 (Rifidi® Edge Server version 3.4)

December 2015

Table of Contents

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, how to export and deploy it.

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 an RFID application that runs on the Rifidi® Edge Server, or reader adapters and plugins for our Edge Server

- The Javadoc for the Rifidi® API can be found online at
  http://www.transcends.co/www/javadoc/edge3.4.0/

- The wiki has some helpful pages to answer some common questions that users and developers have. It can be found at http://wiki.rifidi.net

- The forums provide a way for users and developers to ask questions. It can be found online at
  http://forums.rifidi.net

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.

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 (optional)

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.

- config/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.

Additional Development using Jumpstart Examples

When adding more functionality to your Rifidi® application, check out the Jumpstart examples on our wiki. They can help you accelerate development. The Jumpstart examples can be found at the wiki. Here are some of the key examples:

- Database Jumpstart

  Instead of importing the project template you can import the DB application. Follow the steps described in the Database Jumpstart wiki page , then run the project and set up the configuration files. Finally, export and deploy the DB application.

- JMS Jumpstart

- Cloud Jumpstart

- Rifidi® RFID Services Integration Jumpstart

- Restful Service Integration Jumpstart

- Rifidi® Management API Jumpstart

Follow each Jumpstart's description as outlined on the wiki.

Rifidi® Edge Server How-To's

For a number of developer tasks there are How To's documented that help guide you through a specific task. This guide includes configuration, development, and deployment tasks and is documented on the wiki.

Viewing and Modifying the Rifidi® Source Code

If you desire to change existing Rifidi® Edge Server functionality, you can do this by mofiying its 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.

Build Rifidi® Edge Server from Source

If you would like to build the edge server from source you have two options:

- In a Linux environment by going to the Rifidi-SDK directory and typing "sh build.sh", or

- Any changes you make to the core product (including any additions) can be exported to an OSGi bundle jar, added to the "plugins" folder in a Rifidi install and the "bundles.info". You can find a description how to add OSGi bundles here.

How to build the Rifidi® Edge Server from Source is also described on the wiki.

Best Practice for Designing a Rifidi® Application

When you start out developing a Rifidi® system, consider a best practice approach. We have documented a method outlining steps to building a Rifidi® application on the wiki. The steps include objectives, setup, design, and other topics.

Rifidi® 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). In addition, starting with version 3.1, Rifidi® Edge Server offers an Operations, Administration & Management layer that exposes many internal functions via a restful services interface.

Rifidi Edge Server - Architecture
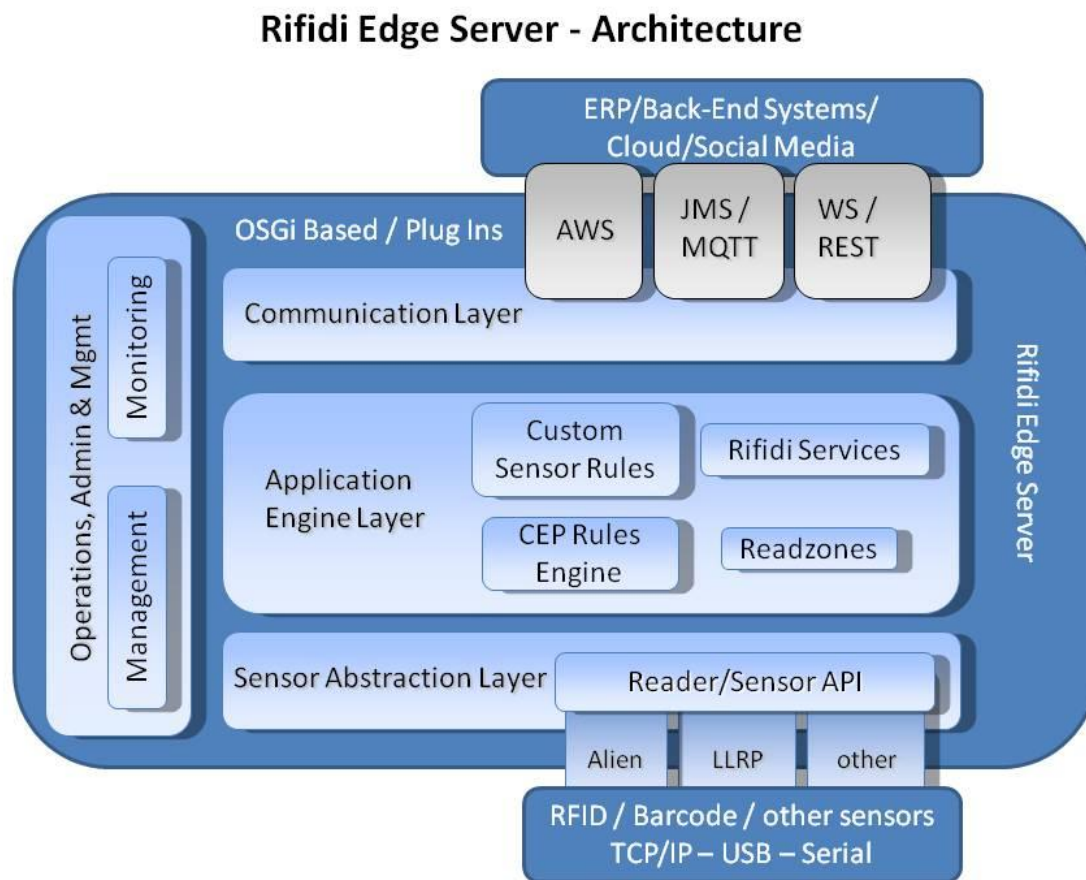
Sensor Abstraction Layer

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, Motorola 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 dependent, it is possible to write your own connector (such as a TCP/IP socket connection) if the application needs it.

Operations, Administration & Management Layer

The purpose of the OA&M layer is to expose Rifidi® internal functions to the developer dynamically at runtime. Many of those functions have been available in prior versions already via RMI, but are now offered via REST Services calls.

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

- Check our new Sensor API document for tips on how to create a sensor adapter.

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 the wiki.

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 supported by the Rifidi® Edge Server starting with version 3.1.

The LLRP Encode Command can be used to encode a new tag with the following operations:

```
Order:  operationCode
 1 - LLRPAccessPasswordWrite
 2 - LLRPEPCWrite
 3 - LLRPKillPasswordWrite
 4 - LLRPEPCLock
 5 - LLRPKillPasswordLock
 6 - LLRPAccessPasswordLock
```

What are EPC, Kill Password, Access Password:

- EPC is the GEN2 EPC ID for a Tag
- Kill Password is used to set the password necessary to kill/destroy a tag for future use (by default there is no password (value(0))
- Access Password is used to perform tag writes such as EPC, Kill Password, Access Password (by default there is no password (value(0))
- C1G2Write and C1G2Lock are the LLRP Access Spec operations used

Command:

- llrpencode/{readerID}/{sessionID}/{tag}
  Performs **all** LLRP Encoding operations for a new RFID Tag.

  Example:
  http://localhost:8111/llrpencode/LLRP_1/1/11112222333344445555666

  Prerequisites:

- Only one tag can be present (this is not a Bulk encoding operation)
- Must be a LLRP reader
- Must be a valid readerID and sessionID
- Tag must be evenly divide by 4 (word in hex) and tag must be able to support length
- Access password and kill password must be evenly divide by 4 (4 or 8 characters/1 or 2 words) or 0
- Only one encoding session per reader can be active at one time

All LLRP Encode operations performed are returned asynchronously to an MQTT Topic if the MQTT variables are defined globally, otherwise synchronous response.

- Global variables:

```
-Dorg.rifidi.llrp.encode.mqttqos=2  -- quality of service level
-Dorg.rifidi.llrp.encode.mqttbroker=tcp://localhost:1883  --  MQTT
broker URL. If set, received response of encode operation is posted to
this queue in asynchronous mode. mqttqos and mqttclientid must be set
too in order to post to queue.
-Dorg.rifidi.llrp.encode.mqttclientid=llrpEncodeId
```

LLRP Encode Response

All LLRP Encode operations performed are returned asynchronously to an MQTT Topic if the MQTT

Success Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
   <message>Success</message>
</response>
```

Fail Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
   <message>Fail</message>
   <Description> LLRP Message returned goes here such as Tag mask and
Tag data not matching </Description>
</response>
```

LLRP Encode Operations Response (if asynchronous)

- Topic naming convention: {readerID}/encode

Success Message:

```
<encodeMessage>
<status>Success</Status>
</encodeMessage>
```

Fail Message:

```
<encodeMessage>
<status>Fail</Status>
<operationList>
  <operation>EPCWrite:Success</operation>
  <operation>KillPasswordWrite:Success</operation>
  <operation>AccessPasswordWrite:Success</operation>
  <operation>EPCLock:No_Response_From_Tag</operation>
  <operation>KillPasswordLock:Success</operation>
  <operation>AccessPasswordLock:Success</operation>
</operationList>
</encodeMessage>
```

LLRP Encode Single Shot Command

These commands are used to perform single operations and override Global properties. All values must be supplied in request. Note: All MQTT (if set), timeout tagmask and epctarget always come from Global Variables.

Command:

- /llrpencode/{readerID}/{sessionID}/{operationCode}/{properties}
  Enables one to call a **single** Encoding operation for a tag.

List of operationCodes:

- LLRPEPCWrite
- LLRPKillPaswordWrite
- LLRPAccessPasswordWrite
- LLRPEPCLock
- LLRPKillPasswordLock
- LLRPAccessPasswordLock

Examples:

- LLRPECWrite - Used to Write a EPC ID

  http://localhost:8111/llrpencode/LLRP_1/1/LLRPEPCWrite/accesspwd=12345678;tag=00000000 0001111111111268

- LLRPKillPasswordWrite - Used to Write a Kill Password
  http://localhost:8111/llrpencode/LLRP_1/1/LLRPKillPasswordWrite/accesspwd=12345678;killpw d=87654321

If MQTT Variables are defined globally then operations result response will be sent to MQTT topic, otherwise response is synchronous.

 For details how to use these features see the [description on the wiki](#).

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.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**

    o **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.

The location of the readzone properties file should be placed in while developing in Eclipse, it is
Rifidi-SDK\RifidiHome\applications\[your application name]\readzones (under your workspace)

Once you export your application, it will be at
applications\[your application name]\readzones (under the server home directory)

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.

Advanced Read Zone Filtering

Starting with version 3.0 Rifidi implements a feature called "Advanced Read Zone Filtering". Using regular expressions a readzone can be defined more flexible, for example as a set of readers. The useRegex flag needs to be set in the Rifidi Application (or a Rifidi Service) and the readzone needs to be configured in the readzone properties file. Below is an example how this feature can be applied, in this case in the readZoneMonitoringService:

- In the application code or Rifidi Service (here: readZoneMonitoringService) the useRegex flag is set to true:

```
this.readZoneMonitoringService.subscribe(weigh_station_subscriber,
weigh_station, this.weighstation_timeout, TimeUnit.SECONDS, true);
```

- In the readzone properties file define the readzone using a regular expression:

```
readerID=reader1|reader2|reader3
```

    In this example, the readzone consists of 3 readers.

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 read zone 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.api.service.tagmonitor.ReadZoneMonitoring
Service"/>
```

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.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.api.service.tagmonitor.UniqueTagBatchInte
rvalService"/>
```

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.api.service.tagmonitor.UniqueTagIntervalS
ervice"/>
```

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.api.service.sensormonitor.SensorStatusMon
itoringService"/>
```

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

```
handleSensorStatusEvent(SensorStatusEvent event)
```

RSSI Monitoring Service

This service monitors which readzone returns the highest average RSSI value for a tag in a given duration. It returns a reader and epc code when the reader with the highest RSSI value changes for the tag over the given timeframe. You can use the following parameters to control the behavior:

- rssiInterval (FLOAT); the duration (in seconds) over which all RSSI values will be accumulated to calculate their average value.

- avgRSSIThreshold (Double); minimum RSSI value in order to be considered for building the average RSSI value. RSSI values below this threshold are ignored. If this value is set to '0', the parameter is disabled.

- tagCountThreshold (Integer); the minimum number of reads for this tag in the given readzone in order to be considered by this service. If the number of reads is lower than the threshold, the reads are ignored. If this value is set to '0', the parameter is disabled.

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

```
<osgi:reference id="rssiMonitoringService"
interface="org.rifidi.edge.api.service.tagmonitor.RSSIMonitoringServ
ice"/>
```

You should implement the RSSIReadZoneSubscriber, which gives you the following method:

```
tagArrived(List<RSSITagReadEvent> incoming)
```

The service returns tag id, antenna, readerid and readzone. In the event readzone properties are not defined or null then readzone will be set equal to readerid (readzone and reader id is by default 1 to 1 relationship).

Note: When using read zones properties file one will need to be sure all reader ids map to a read zone otherwise there is a chance a tag will not be returned for an undefined reader id. One best practice is to create a default readzone where you exclude through a regular expression all readzones which will be specifically defined. An Example regex in readzone property file is readerid=^((?!reader1|reader2).)*$. This will exclude reader1 and reader2 from the default read zone and then more specific read zones properties can be defined for the readers such as associating multiple readzones to a reader by antenna (1 to many relationship) or multiple readers correlated to same readzone (regex: reader1|reader2) (many to 1 relationship).

How to add a new Service

In order to add a new Service to the Rifidi® Edge Server core, you need to follow the following steps:

1. Add your classes to the package "org.rifidi.edge.api.service.tagmonitor"

2. Add an entry like this to app-api.xml (a spring XML):

```
<bean_id="RSSIMonitoringService"
class="org.rifidi.edge.api.service.tagmonitor.RSSIMonitoringServiceI
mpl"><constructor-arg index="0" value="AppService"/><constructor-arg
index="1" value=RSSI"/></bean>
```

Give the name you want to the bean ID, and give your implementation class as the class. Constructor arg 0 must be "AppService". Constructor arg 1 can be anything that doesn't already exist.

3. Add an entry like this to app-api.xml:

```
<osgi:service ref="RSSIMonitoringServie" auto-export="interfaces"/>
```

The ref should be your bean ID.

After that, your service should be available. You can find a description also on the [wiki](#).

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 has 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"/>
```

Edge Messaging - MQTT

Prior to Rifidi® Edge Server 3.1 Edge Messaging is available through an Active/JMS interface. Starting with version 3.1 Edge Messaging is now accessible through MQTT, a light weight IoT messaging protocol through leveraging Moquette. This now enables Rifidi applications to publish events (captured via Rifidi Services for example Readzone Monitoring, Stable Set, etc.) to a light weight MQTT messaging queue.

Moquette start by default (see rifidiserver.ini):

```
-Dorg.rifidi.mqtt.enabled=true
```

For additional details and examples see the description on the Wiki.

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. We give also a brief explanation on the wiki and explain steps below.

How to connect Rifidi via RMI Client

1. Ensure you set client java property to ensure ssl Cipher matches
    javax.rmi.ssl.client.enabledCipherSuites=SSL_DL_anon_WITH_RC4_128_MD5

2. Sample RMI Client code

```
RmiProxyFactoryBean proxy = new RmiProxyFactoryBean();
proxy.setCacheStub(true);
proxy.setRefreshStubOnConnectFailure(true);
proxy.setServiceInterface(SensorManagerService.class);
proxy.setServiceUrl("rmi://[replace with ipaddress such as
127.0.0.1]:1101/SensorManagerService");
proxy.setRemoteInvocationFactory(new
ContextPropagationRemoteInvocationFactory());
proxy.afterPropertiesSet();
SensorManagerService service = (SensorManagerService)
proxy.getObject();
ReaderDTO reader = service.getReader("LLRP_1");
}
```

Amazon AWS

Rifidi® now has support for the Amazon AWS. To learn more about how to connect, please check our documentation at the wiki.

Operations, Administration & Management Layer

The OA&M section describes how internal functions allowing the management and monitoring of the edge server and its components are exposed via restful services and how those can be used by a developer.

Management

Prior to version 3.1 Edge Management has been available only through RMI (currently used by Rifidi® Workbench, if you are looking for example code). Starting with version 3.1, Edge Management is now also accessible through Restful Services (leveraging Restlet plugin). Restlet allows all Rifidi® Edge Management operations that are available through workbench to be accessible via Restful Services.

The types of operations available through Restful Services include:
*   All "session" commands (for stopping and starting reader/sensor sessions)

- ExecuteCommand and deleteCommand (for commands supported by a sensor such as a tag read)
- Readers (for getting a list of readers/sensors available on Edge Server instance)
- Commands (for issuing command line operations such as saving the Edge Server configuration currently in memory)
- Get and Set Properties (for setting and getting sensor properties such as Setting the LLRP Reader Configuration dynamically)
- Create Reader (for creating a new reader connection)
- Managing Rifidi Applications (for stopping/starting, listing and deploying Rifidi® applicatoins)

For more examples and additional details please read the documentation on the wiki, or the jumpstart example here.

Monitoring

Prior to version 3.1, Edge Monitoring has been available only through JMX/MBeans via RMI. Starting with version 3.1 Edge Monitoring is now also accessible through Restful Services. Rifidi® leverages Jolokia which exposes all JMX/Mbeans via restful services.
The types of components which can be monitored include:
- Java Virtual Machine (Memory, CPU etc.)
- Sensors/Readers (Status, Properties etc.)
- OSGi (Status, Applications etc.)
- Messaging

For examples and additional details please read the documentation on the wiki.

Enterprise Rifidi® Management Dashboard

The Web Administration Dashboard enables users to manage, monitor and configure Rifidi Servers, Sensors/Readers and Applications.The key components of the enterprise-wide dashboard that was added with release 3.3 are summarized below.

- Edge Server Node Management – Let's you monitor multiple Rifidi Edge server nodes across the enterprise to see connection states, properties, sensors and applications.
- Sensor Management – Provides an interface to monitor sensors' state, manage sensors' properties and dynamically update sensors' execution.
- Rifidi Application Management – Allows you to monitor application states, manage sensor/reader read zones, and update both out-of-the-box business solutions and custom (via SDK) Rifidi sensor/IoT device properties, providing the foundation for a Rifidi Application marketplace.

For more information on the dashboard read the documentation on the wiki.

ThingWorx IoT Platform Integration

As part of the ThingWorx-Ready Program Rifidi® is enabled to integrate with the ThingWorx Platform.

For more information and a jumpstart application please read the documentation on the wiki.

Important: In order to use this ThingWorx Integration jumpstart application or any ThingWorx Platform Integration/Use cases you will need to contact ThingWorx to setup an account - wwww.thingworx.com

Exporting and Deploying

For help in exporting and deploying a Rifidi®App, follow this 4-step process:

- Step 1: Export the Application Bundles

- Step 2: Create the Application

- Step 3: Deploy the Applications

- Step 4: Starting application by default when Rifidi® Edge Server starts.

The full description can be found on the wiki.

Performance Tuning

For help with improving the performance of Rifidi® Edge Server look at the Performance Tuning Tips on the Wiki. Those tips include descriptions how to turn off diagnostic and monitoring functionality, adjusting configuration settings for performance, Esper and other useful tips.

Examples

For developers who are new to Rifidi® Edge Server there are several Development Jumpstarts documented on the wiki. We are introducing some of them in the following sections. Here is a full list of them:

- HelloWorld App Jumpstart

- Database Application Jumpstart

- MQTT Jumpstart

- AWS / Cloud Jumpstart

- Rifidi Management API Jumpstart

- Dynamic Reader Configuration Jumpstart

- Rifidi Services Jumpstart

- Northwind Application

- Cloud

HelloWorld Application

The HelloWorld application is a very simple Rifidi® application. It uses the ReadZoneMonitoringService to create a subscriber, and prints tag epc id to standard out when a tag arrives in the read zone. Please see the description on the wiki for more details.
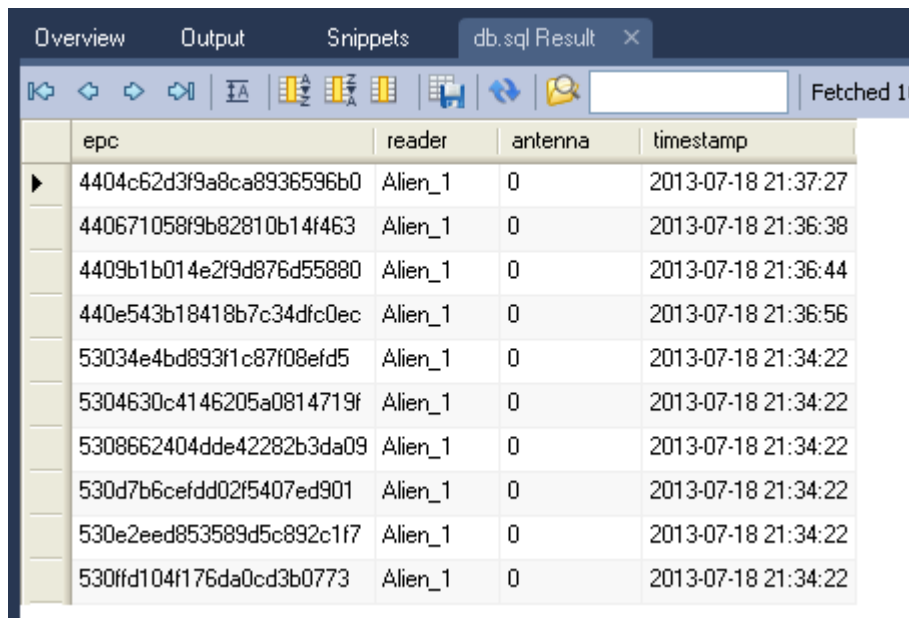
Database Application

You just purchased an RFID reader and a few tags and are wondering how Rifidi® can help you make use of your new hardware. A common use case developers want to accomplish is to record when a tag shows up, and what reader or antenna is "seeing" it, and record this in a database. Our simple DB App example is doing just that. It generates exactly one entry every time a tag enters the read zone of a reader and records the timestamp and reader ID.

In order to try out the DB app you can just download the application and the sql file from here, then install the app and run the sql file to create the table in your database.

If you want to modify the DB app, then download it from here, and import it to your Rifidi® development environment. You can find all the required steps on our wiki.

The table structure will look like:

To summarize, here are the components you need:

- Rifidi® Edge Server, with the application DBApp and the DBApp.properties file.

- MySQL database, with the db schema installed. Use db.sql file to generate it.
Default values for the database:

  dbapp.url=jdbc:mysql://127.0.0.1/db, userr=root, pass=rifidi
  You can override these values in the .ini for windows or the "rifidi-server" file for Linux if you wish.

Northwind Application

After your DB app is running, you could consider a more sophisticated, but also complex application. The Northwind application example will introduce you to writing custom esper rules. Please go to our Northwind example on the wiki to find details on this application. However, we do recommend when you build esper rules, to build them as part of a service as shown in the chapter RIFIDI® Services.

Dynamic Reader Configuration Example

In some situations you might want to update your reader configuration dynamically, during run-time, based on some business or infrastructure events. This specific example application updates an existing reader's LLRP RoSpec configuration in run-time. This could be extended to update other reader properties or configurations dynamically. The example app can be found on the wiki.

Esper is a trademark of EsperTech Inc. All other trademarks, product names, and company names or logos cited herein are the property of their respective owners.