
Elixir Repertoire Runtime

Elixir Technology Pte Ltd

Copyright © 2009 Elixir Technology Pte Ltd

All rights reserved.

Solaris, Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. Microsoft and Windows are trademarks of Microsoft Corporation.

Table of Contents

Elixir Repertoire Runtime	1
Overview	1
Getting Started	1
Repertoire Runtime API	1
Using the Runtime API	1

Elixir Repertoire Runtime

Overview

The Elixir Repertoire Runtime is a data and report generation engine that can read Elixir datasources and report templates and render the results into all Elixir supported formats from within your own Java application. The goal of the runtime is to provide a simple and straightforward Application Programming Interface (API) for developers to access data generation and reporting functionality such as previewing/printing and saving of reports.

Getting Started

The Java Virtual Machine (JVM) platform required to run the Runtime Engine is Java 5 or greater. In order to develop Java applications you will need the corresponding Java Development Kit.

The runtime module is located in the runtime subdirectory in the location where you installed Elixir Repertoire.

Note

There is a Readme.html in the runtime directory which contains additional information.

Repertoire Runtime API

Using the Runtime API

The Repertoire Runtime API allows the you to programmatically add report and data source functionalities to your Java application. Using a the combination of the API allows the creation of all kinds of Java application, including standalone GUIs, batch-mode generators and Servlets.

Elixir tools use the Apache Foundation's Log4j for logging information and error messages. Your application should configure log4j before initializing the runtime. The easiest way to do this is to add the log4j jar file to your classpath, import the BasicConfigurator class and use:

```
// import org.apache.log4j.BasicConfigurator;
BasicConfigurator.configure();
```

Initializing the Runtime

The class `ReportEngineFactory` provides the functions to setup the Runtime. The steps to use this class are:

- Initialize the `ReportEngineFactory` by calling `init`. The interface `IReportHost` allows you to implement your own class to log messages from the runtime. The `BasicReportHost` is the default implementation provided. The code example is shown below.

```
ReportEngineFactory.init(new BasicReportHost());
```

- As the Runtime supports multi threading, you may control the amount of threads to render reports concurrently and the number of thread to wait in the queue. As Report generation is a very expensive process, the amount of report that can renders concurrently is dependent on the system resources. The report rendering threads and number in queue is in turned controlled by the licensing. It will not exceed this amount. The default runtime license sets the number of threads to one. Please contact Elixir to obtain licenses for multi-threaded use. If the setup permit permits two report rendering threads and five threads to wait in queue. If the user triggers ten report generation requests, two reports will be generated concurrently, five report rendering threads will wait in queue and the rest of the requests will be rejected.

```
IConfigurator conf = ReportEngineFactory.getConfigurator();
if (conf instanceof IRuntimeConfigurator)
{
    IRuntimeConfigurator rconf = (IRuntimeConfigurator)conf;
    rconf.setMaxRenderCount(2);
    rconf.setMaxQueueCount(5);
}
```

- The next step is to obtain the Report Administration interface to setup the repository location. This is done by using `loadRepository`. This method loads up the repository configuration file which defines the filesystems and their properties. You can use a physical config file location or an inputstream to locate the configuration. The name of this configuration file is typically `repository.xml`.

The following illustrate the possible use of `loadRepository`:

```
IReportAdmin reportAdmin = ReportEngineFactory.newAdminInstance();
File repXML = new File("C:\\Config\\repository.xml");

// 1/ Load via a physical configuration file
reportAdmin.loadRepository(repXML);

// OR

// 2/ Load via a Java Input Stream
FileInputStream fis = new FileInputStream(repXML);
reportAdmin.loadRepository(fis);
```

The alternate approach to loading filesystems into the repository is to use `addJarFileSystem` for jar files or `addLocalFileSystem` for physical file path location. Below is an example of the usage:

```
reportAdmin.addLocalFileSystem("ReportSamples",
    "C:\\\\Demo\\\\sample-reports");
```

- Having setup the repository, you can get an instance of the Engine using `newEngineInstance()`.

```
IReportEngine engine = ReportEngineFactory.newEngineInstance();
```

- The API, `getFileSystems` retrieves the file systems in the repository and `getReportNames()` is to list the reports available in that filesystem.

```
IReportEngine engine = ReportEngineFactory.newEngineInstance();

// 1/ Listing the filesystems available in the repository
String [] filesystems = engine.getFileSystems();

// 2/ Listing reports deployed in filesystem.
String [] reportNames = engine.getReportNames("MyFileSystem");
```

Using Runtime's Report API

The report API allows you to list filesystems, list reports deployed in the file system and generate report output in different formats such as pdf, excel etc.. The codes below illustrate how the APIs may be used.

```
IReportEngine engine = ReportEngineFactory.newEngineInstance();
File outPDF = new File("C:\\\\MySaleReport.pdf");
FileOutputStream os = new FileOutputStream(outPDF);

//Set dynamic report parameters into Properties
Properties props = new Properties();
props.setProperty("ID", "1234");

//get a report object
RawReport raw = engine.getRawReport("/MyFileSystem/SaleReport.rml");
//Generate the report as pdf file document.
//A JobInfo object is returned to give a status.
IJobInfo jobinfo = engine.renderReport(raw, "application/pdf",
    os, props);
```

Using Runtime's DataSource and DataStore API

The datasource API allow you to interface with the Data store either to extract a set of data out from the the data source. This is useful to verify information or to trigger the DataStore to push data into a source. The later requires the appropriate license to permit this activity.

```
File outXLS = new File("C:\\\\MySaleData.xls");
OutputStream outputStream = new FileOutputStream(outXLS);

//Set dynamic report parameters into Properties
Properties props = new Properties();
props.setProperty("ID", "1234");
//extract the data back as excel spread sheet.
props.setProperty("mime-type", "application/vnd.ms-excel");

// Get Data source.
```

```
IJobInfo jobinfo = engine.generateData("/MyFileSystem/SaleData.ds",  
    outputStream, props);
```

The data store can be triggered by the same API except a new property has to be passed in, the data store name. The key name is datastore.

```
Properties props = new Properties();  
//Choose the data store to trigger in the composite datasource  
props.setProperty("datastore", "MyJDBCStore");  
reportEngine.generateData("/MyFileSystem/MyCompositeDS.ds",  
    outputStream, props);
```