Seam Reports Module

Reference Guide

by George Gastaldi and Jason Porter

Introduction

The goal of Seam Reports is to provide a fully integrated CDI programming model portable extension for Java EE that provides APIs for compiling, populating and rendering reports from existing report frameworks.

Seam Reports contains similar functionality to that of the Excel and PDF templates of Seam 2, however, the creation and compilation of the reports is quite different. Seam Reports aligns much better with existing tools in a business, making use of knowledge and expertise that exists outside of development. The functionality in Seam 2 was largely targeted to creating flyers and simple pages. Seam Reports can accomplish this, but also allows for easy creation of multi-page business reports by integrating with *JasperReports* [http://jasperforge.org/projects/jasperreports], *Pentaho* [http://www.pentaho.com/], and *XDocReports* [http://code.google.com/p/xdocreport/]. Integration with other reporting solutions can also be done easily by implementing five small interfaces provided by Seam Reports, see chapter 3 for more information about adding reporting engines.

Installation

Most features of Seam Reports are installed automatically by including the seam-reports-api.jar and the respective provider implementation (along with its dependencies) in the web application library folder. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your pom.xml file:

```
<dependency>
   <groupId>org.jboss.seam.reports</groupId>
   <artifactId>seam-reports-api</artifactId>
   <version>${seam-reports-version}</version>
</dependency>

<!-- If you are using Jasper Reports, add the following dependency -->
<dependency>
   <groupId>org.jboss.seam.reports</groupId>
   <artifactId>seam-reports-jasper</artifactId>
   <version>${seam-reports-version}</version>
</dependency>

<!-- If you are using Pentaho, add the following dependency -->
<dependency>
   <groupId>org.jboss.seam.reports</groupId>
   <artifactId>seam-reports-pentaho</artifactId>
   <version>${seam-reports-version}</version>
</dependency>
```

Tip

Replace ${seam-reports-version} with the most recent or appropriate version of Seam Reports.

1.1. Installation using Seam Forge

If you are using Seam Forge, you may use the seam-reports plugin to help with the setup.

1.1.1. Plugin Installation

If not already installed yet on Forge, you may install the plugin by running the following command inside Forge:

```
forge git-plugin git://github.com/forge/plugin-seam-reports.git
```

1.1.2. Plugin Configuration

- To add only the api:

```
seam-reports setup
```

- To configure Seam Reports to work with JasperReports:

```
seam-reports setup --provider JASPER
```

- To configure Seam Reports to work with Pentaho Reporting Engine:

```
seam-reports setup --provider PENTAHO
```

Usage

2.1. Quick Start

Using Seam Reports is a simple four step process. These steps are the same regardless of the reporting engine being used.

1. Create a report using a favorite report editor
2. Load the created report
3. Fill the report with data
4. Render the report

Of course some of these steps will have different ways of accomplishing the task, but at a high level they are all the same. For simplicity this quick start will use JasperReports and the first step will be assumed to have already taken place and the report is available in the deployed archive. The location of the report isn't important, the ability to pull it into an `InputStream` is all that really matters.

The following code demonstrates a basic way of fulfilling the last three steps in using Seam Reports using JasperReports as the reporting engine. The report has already been created and is bundled inside the deployable archive. There are no paramaters for the report. The report is a simple listing of people's names and contact information.

```
@Model
public class PersonContactReport {
  @Inject @Resource("WEB-INF/jasperreports/personContact.jrxml")          (1)
  private InputStream reportTemplate;

  @Inject @Jasper                                                         (2)
  private ReportCompiler reportCompiler;

  @Inject @Jasper @PDF                                                    (3)
  private ReportRenderer pdfRenderer;

  @Inject
  private EntityManager em;
                                                                          (4)
  public OutputStream render() {
    final Report filledReport = this.fillReport();
    final OutputStream os = new ByteArrayOutputStream();
```

```
    this.pdfRenderer.render(filledReport, os);
    return os;
  }

                                                    (5)
  private Report fillReport() {
    final ReportDefinition rd = this.reportCompiler.compile(reportTemplate);
    return rd.fill(this.createDatasource(), Collections.EMPTY_MAP);
  }

                                                    (6)
  private JRDataSource createDatasource() {
              final List<Person> personList = this.em.createQuery("select p from
Person", Person.class).getResultList();
    return new JRBeanCollectionDataSource(personList);
  }
}
```

(1) Solder allows easy resource injection for files available in the archive. This injects the report template which has been created previously (perhaps by someone else in the business) and added to the deployable archive.

(2) A `ReportCompiler` is an interface from Seam Reports which abstracts compiling the report template into `ReportDefinition`. Seam Reports makes use of CDI's type safety features by using qualifiers to further narrow the intended type. This allows programs to remain implementation agnostic. The `Jasper` qualifier annotation instructs CDI to inject an implementation of the `ReportCompiler` which contains the same qualifer.

(3) This is an instance of using both a qualifer (`@Jasper`) and also a metadata annotation, which happens to be a stereotype. The `@PDF` annotation is a CDI stereotype, which essentially means it's a group of other annotations. It carries metadata about the type it is decorating. More about this later.

(4) The `render` method is the only entry point into the class, it also returns the final output of generating a report. It makes use of other methods in the class to finish the steps outlined above to generate a report using Seam Reports.

(5) At this stage data to populate the report is retrieved and added to the compiled `ReportDefinition`. This particular report doesn't make use of any parameters, hence the empty map instance being passed.

(6) This last stage of using Seam Reports is the only place that may require the application to use the report engine API. In this example a list of JPA entities is retrieved and added to a JasperReports datasource, which is then used by the calling method to populate the report template as mentioned above.

2.2. Annotations

There are four API level annotations to be aware of when using Seam Reports. All four of them declare metadata about objects that are being injected. They're also all CDI stereotypes which instruct the implementing renderer the mimetype that should be used.

- CSV
- PDF
- XLS
- XML

These annotations are only used when injecting a `ReportRenderer`. Only one of them may be used per renderer. Multiple renderers must be injected if multiple renderering types are desired.

2.3. Troubleshooting