Juzu Web Framework
0.4.3
Tutorial

Julien Viet

eXo Platform

Table of Contents

Preface

Juzu is a web framework based on MVC concepts for developing Portlet applications. Juzu is an open source project developed on GitHub project licensed under the LGPL 2.1 license.

This tutorial will make you familliar with Juzu, to reach our objective we will develop a weather application in several steps, each step introducing a new feature to gradually improve the application.

1

Quickstart

1.1. Deploy the applications

Before diving in the technical part of this tutorial, we need to study how to deploy the examples and how to use them. In the package you downloaded you will find a war file adapted to your portal server in the `/tutorial` directory:

- `juzu-tutorial-examples-gatein.war` for the GateIn portal server

- `juzu-tutorial-examples-liferay.war` for the Liferay portal server

The main reason we have two servers is that the jars are not exactly the same, each is adapted to the portal server you will use. When you deploy the applications, the deployment process will print information in the console, similar to:

```
INFO: Deploying web application archive juzu-tutorial-gatein.war
[Weather4Portlet] Building application
[Weather4Portlet] Using injection org.juzu.impl.spi.inject.cdi.CDIBootstrap
[Weather4Portlet] Starting Weather4Application
[Weather4Portlet] Dev mode scanner monitoring /java/gatein/webapps/juzu-tutoria
[Weather3Portlet] Building application
[Weather3Portlet] Using injection org.juzu.impl.spi.inject.spring.SpringBootst
[Weather3Portlet] Starting Weather3Application
[Weather3Portlet] Dev mode scanner monitoring /java/gatein/webapps/juzu-tutoria
[Weather5Portlet] Building application
[Weather5Portlet] Using injection org.juzu.impl.spi.inject.cdi.CDIBootstrap
[Weather5Portlet] Starting Weather5Application
[Weather5Portlet] Dev mode scanner monitoring /java/gatein/webapps/juzu-tutoria
[Weather2Portlet] Building application
[Weather2Portlet] Using injection org.juzu.impl.spi.inject.cdi.CDIBootstrap
[Weather2Portlet] Starting Weather2Application
[Weather2Portlet] Dev mode scanner monitoring /java/gatein/webapps/juzu-tutoria
[Weather1Portlet] Building application
[Weather1Portlet] Using injection org.juzu.impl.spi.inject.cdi.CDIBootstrap
[Weather1Portlet] Starting Weather1Application
[Weather1Portlet] Dev mode scanner monitoring /java/gatein/webapps/juzu-tutoria
```

As we can notice, there are 5 applications deployed, one for each of the topic of this tutorial

- Weather1Application: Chapter 1, *Quickstart*

- Weather2Application: Chapter 2, *Template overwiew*

- Weather3Application: Chapter 3, *Dependency Injection*

- Weather3Application: Chapter 4, *Views*

- Weather3Application: Chapter 5, *Actions*

1.2. Interacting with the application

In this tutorial, Juzu applications are deployed in the *dev* mode. This runtime mode allows you to modify the source code of the application, Juzu will pick up the modifications and update the running application almost instantanesouly.

The source code for the five applications is in the `/WEB-INF/src` directory of the war file, each application has its own package, for instance the quickstart application uses the package `examples.tutorial.weather1`. The first version of the application shows the most basic Juzu application. Our application is declared in the `examples.tutorial.weather1` package package annotated with the `@Application` annotation This annotation declares a Juzu application and does not require any mandatory value. Like classes, methods or fields, Java packages can be annotated, such packages declaration are represented by a special file named `package-info.java`.

Usually an application is made of controllers and templates, in this example, the `Weather` Java class contains a method annotated with the `@View` annotation, which turns the `Weather` class into a Juzu controller. The controller method `index()` is the name of the default method that Juzu will call.

```java
@View
public void index()
{
    index.render();
}
```

Methods annotated by `@View` have the unique purpose of providing markup, they are called *view*. In our case, the method delegates the rendering to the `index.gtmpl` template. The template is injected in the controller thanks to the `@Inject` annotation and the `@Path("index.gtmpl")` annotation.

```java
@Inject
@Path("index.gtmpl")
Template index;
```

By default templates are located in the `templates` package of the application, in our case the `examples.tutorial.weather1.templates` package. The `@Path` annotation specifies the path of the template in this package. The templates are located in the same source tree than the java classes because the files must be available for the Java compiler.

2

Template overview

Now we will improve our application by exploring a bit the templating engine. We will show a quick overview of Juzu templating system. Templates are essentially made of static part (usually markup) and dynamic parts. In this section we will focus on explaining the use of dynamic expression in a template.

The application shows how a view can provide variable input for a dynamic template with parameters. Our application has a view controller and a template, but now the template contains the `${ }` expression that makes it dynamic.

```
The weather temperature in ${location} is ${temperature} degrees.
```

Like before the template is used in the view controller but now we use a `Map` containing the `location` and `temperature` parameters.

```java
@View
public void index()
{
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("location", "Marseille");
    parameters.put("temperature", "20");
    index.render(parameters);
}
```

During the template rendering, the `location` and `temperature` expressions are resolved to the value provided by the view controller. When a template is rendered, an optional map can be provided, this map will be available during the rendering of the template for resolving expression.

3

Dependency Injection

The next step is to make our application obtain real data instead of the hardcoded values we used in the previous section. For this matter we use a remote service that we encapsulate into the `WeatherService`.

```java
public class WeatherService
{

   private final XPathExpression xpath;

   public WeatherService() throws XPathException
   {
      xpath = XPathFactory.newInstance().newXPath().compile("//temp_c/@data");
   }

   public String getTemperature(String location)
   {
      try
      {
         String url = "http://www.google.com/ig/api?weather=" + location;
         InputSource src = new InputSource(url);
         src.setEncoding("ISO-8859-1");
         return xpath.evaluate(src);
      }
      catch (XPathExpressionException e)
      {
         return "unavailable";
      }
   }
}
```

Juzu uses dependency injection to interact with a service layer. The JSR-330, also knowns as `@Inject`, defines an API for dependency injection. The `WeatherService` is injected in the controller with the `weatherService` field annotated with the `@Inject` annotation:

```java
   @Inject
   WeatherService weatherService;
```

This service is then simply used into our controller `index()` method:

```java
@View
public void index()
{
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("location", "Marseille");
    parameters.put("temperature", weatherService.getTemperature("marseille")
    index.render(parameters);
}
```

As we can see, Juzu relies on the portable `@Inject` annotation to declare sinjections. Injection is performed by the dependency injection container. At the moment the following containers are supported:

- Spring Framework

- JBoss Weld

There is a preliminary support for Google Guice 3.0, but it is not yet available. In the future more container support could be achieved.

By default it uses the *Weld* container, if you want instead to use *Spring* container instead the configuration is done by a portlet init param defined in the deployment descriptor of the portlet:

```xml
<init-param>
  <name>juzu.inject</name>
  <value>spring</value>
</init-param>
```

In the case of *Spring*, the file `spring.xml` file is needed, it contains the service declarations for the Spring container.

Juzu provides more advanced dependency injection, in particular it uses the `Qualifier` and `Scope` features defined by the JSR-330 specification that will be studied later.

Views

Until now we have seen a basic view controller, in this section we will study more in depth view controllers. A view controller is invoked by Juzu when the application needs to be rendered, which can happen anytime during the lifecycle of an application.

This version has still the `index()` view controller, but now it has also an overloaded `index(String location)` method that accept a `location` argument as a view parameter.

```java
@View
public void index(String location)
{
   Map<String, Object> parameters = new HashMap<String, Object>();
   parameters.put("location", location);
   parameters.put("temperature", weatherService.getTemperature("marseille")
   index.render(parameters);
}
```

View parameters are bound to the current navigation of the application and their value are managed by the framework. At this point it is normal to wonder how a view parameter value can change. Let's have a closer look at the `index.gtmpl` application template.

```
The weather temperature in ${location} is ${temperature} degrees.
<a href="@{index(location = 'marseille')}">Marseille</a>
<a href="@{index(location = 'paris')}">Paris</a>
```

The template now has two links that change the view parameters when they are processed. The links are created by a special syntax that references the view method, for instance the script fragment `@{index(location = 'paris')}` generates an url that updates the `location` view parameter to the `paris` value when it is processed.

The initial controller method `index()` is still there but now it simply invokes the `index(String location)` controller with a predefined value.

```
@View
public void index()
{
    index("marseille");
}
```

We could't close this section without talking a bit about **safe urls**. Juzu is deeply integrated at the heart of the Java compiler and performs many checks to detect applications bugs during the application compilation. Among those checks, templates are validated and the url syntax @{ } is checked against the application controllers. In fact Juzu will resolve an url syntax until it finds one controller that resolves the specified name and parameters. If not Juzu will make the compilation fail and give detailled information about the error. This kind of feature makes Juzu really unique among all other web frameworks, we will see some other later.

Juzu leverages the Annotation Processing Tool (APT) facility standardized since Java 6. APT works with any Java compiler and is not specific to a build system or IDE, it just works anywhere, we will see later that it even works with Eclipse incremental compiler.

5

Actions

Now it's time to introduce action controllers, actions are method annotated by the `@Action` annotation. Unlike views, actions are only called when an action url is processed by the portal, whereas a view controller method can be invoked any time by the portal.

The role of an action controller is to process actions parameters. Each parameter of an action controller method is mapped to the incoming request processed by the portal, such parameters can be encoded directly in the URL or be present in the form that triggers the action.

```
The weather temperature in ${location} is ${temperature} degrees.

<ul><% locations.each() { location -> %>
<li><a href="@{index(location = location)}">${location}</a></li>
<% } %>
</ul>

<form action="@{add()}" method="post">
   <input type="text" name="location" value=""/>
   <input type="submit"/>
</form>
```

In our example, we use a form which contains the the #location# action parameters. In order to create an action url we use the same syntax shown for view url `@{add()}` but this time we don't need to set any parameter, instead the form parameters will be used when the form is submitted. However this is not mandatory and instead we could have url parameters such as `@{add(location = 'washington'}`, such syntax is valid specially when it is used without a form. Obviously there is the possibility to mix form and action parameters.

When the url is processed, the following action controller method will be invoked:

```
@Action
public Response add(String location)
{
   locations.add(location);
   return Weather_.index(location);
}
```

The method process the `location` parameter and add it to the `locations` set. After this the

portal will proceed to the page rendering phase and will call the `index()` method to refresh the application.

6

Type safe templating

We have seen previously how render templates from a controller by passing them parameters. Templates use ${ } expressions that often refers to parameters passed by the controller when it renders the template. For this purpose we used an `HashMap` in which we put the various parameters that the template will use during rendering.

This syntax is a generic way to do by using an untyped syntax, indeed if a template parameter name changes the controller will continue to compile because of the generic parameter map. To improve this situation, parameters can be declared thanks to a `param` tag inside the template:

```
#{param name=location/}
#{param name=temperature/}
#{param name=locations/}

The weather temperature in ${location} is ${temperature} degrees.

<ul><% locations.each() { location -> %>
<li><a href="@{index(location = location)}">${location}</a></li>
<% } %>
</ul>

<form action="@{add()}" method="post">
   <input type="text" name="location" value=""/>
   <input type="submit"/>
</form>
```

For instance the `location` parameter is declared by the `#{param name=location/}` tag. During the Java compilation, Juzu leverage those parameter declarations to provide a more convenient way to render a template.

Indeed the tight integration with the Java compiler allows Juzu to generate a template class for each template of the application, such template class inherits the `Template` class and adds specific methods for passing parameters to a template in a safe manner.

```java
@View
public void index(String location)
{
    index.location(location).
            temperature(weatherService.getTemperature(location)).
            locations(locations).
            render();
}
```

As we can see, the `HashMap` is not used anymore and now we use a type safe and compact expression for rendering the template. Each declared parameter generates a method named by the parameter name, for the `location` parameter, we do have now a `location(String location)` method that can be used. To make the syntax fluent, the parameter methods can be chained, finally the `render()` method is invoked to render the template, however it does not require any parameter since all parameters where passed thanks to the parameter methods.

The Java name of the generated template class is the name of the template in the `templates` package of the application. In our case we do obtain the `examples.tutorial.weather6.templates.index` class name. It is very easy to use our subclass by injecting the template subclass instead of the generic `Template` class.

```java
@Inject
@Path("index.gtmpl")
examples.tutorial.weather6.templates.index index;
```

Of course it is possible to import this value and use directly the `index` class name. We used directly the full qualified name of the class for the sake of the clarity.

7

Wrap up

We reached the ends our walk through Juzu, now you can learn more and study the Booking application. This application can be found in the package you downloaded in the `booking` directory.