Table of Contents

Preface

1. Quickstart
   - 1.1. Deploy the applications
   - 1.2. Interacting with the application

2. Template overwiew

3. Dependency Injection

4. Views

5. Actions

6. Type safe templating

7. Styling the application
   - 7.1. The style
   - 7.2. Plugins in action

8. Adding Ajax
   - 8.1. Javascript to the rescue
   - 8.2. Bridge the gap with Ajax

9. Testing our application
   - 9.1. Setting up the test
   - 9.2. Testing the app

10. Wrap up

List of Examples

3.1. Using Spring IOC in a servlet
3.2. Using Spring IOC in a portlet
7.1. The necessary Bootstrap less files
7.2. Injecting Bootstrap CSS in our application
9.1. Using the Arquillian runner
9.2. Application deployment
9.3. Arquillian injection
9.4. Creating URL for an application
9.5. Creating URL for an application

Preface

Juzu is a web framework based on MVC concepts for developing 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-tomcat.war` for the Tomcat server
- `juzu-tutorial-examples-gatein.war` for the GateIn portal server
- `juzu-tutorial-examples-liferay.war` for the Liferay portal server

> In Tomcat server the application is executed as a Servlet whereas in GateIn or Liferay, the application is executed as a Portlet.

The main reason we have several servers is that the jars are not exactly the same, each is adapted to the 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-tomcat.war
[Weather1Portlet] Using injection CDI_WELD
[Weather1Portlet] Building application
[Weather1Portlet] Starting Weather1Application
[Weather2Portlet] Using injection CDI_WELD
[Weather2Portlet] Building application
[Weather2Portlet] Starting Weather2Application
[Weather3Portlet] Using injection INJECT_SPRING
[Weather3Portlet] Building application
[Weather3Portlet] Starting Weather3Application
....
```

As we can notice, there are 8 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*
- Weather4Application: Chapter 4, *Views*
- Weather5Application: Chapter 5, *Actions*
- Weather6Application: Chapter 6, *Type safe templating*
- Weather7Application: Chapter 7, *Styling the application*
- Weather8Application: Chapter 8, *Adding Ajax*

1.2. Interacting with the application

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`.

The first thing to do when developping a Juzu application is to declare the application. The package of the application must be annotated with the `@juzu.Application` annotation to declare the application. The Java file *examples/tutorial/weather1/package-info.java* contains the package declaration along with the annotation:

```
@Application
@Route("/weather1")
@Portlet
package examples.tutorial.weather1;
```

Along with the `@Application` annotation there are two other annotations `@Route` and `@Portlet` :

- The `@Route` annotation defines the route of the application when the application is used as a servlet. It binds the application to the */weather1* path.
- The `@Portlet` annotation generates a portlet application class used by the portlet container

Both annotations are not coupled, you can use either the `@Route` or the `@Portlet` annotation, or both if you want tod deploy your application in both runtimes.

This is enough to create an empty Juzu application, now let's see the application itself!

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.

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

Methods annotated by `@View` have the unique purpose of providing markup, they are called *views*. 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.

```
@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.

The last step of this section is to explain how to package the application, let's see how to do that.

1.2.1. Packaging the application in Tomcat

We need to provide a *web.xml* descriptor:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun
         version="3.0">
  <context-param>
    <param-name>juzu.run_mode</param-name>
    <param-value>prod</param-value>
  </context-param>
</web-app>
```

There are two servlets used for serving the application:

- The `JuzuServlet` serves the Juzu applications contained in the war file
- The `AssetServlet` serves the asset of the applications such as stylesheets or JavaScript

1.2.2. Packaging the application for the GateIn portal

Our application is annotated with the `juzu.plugin.portlet.Portlet` annotation. The `@Portlet` annotation generates a Java class `examples.tutorial.weather1.Weather1Portlet` that we specifies in the *WEB-INF/portlet.xml* deployment descriptor of the web application:

```
<portlet>
  <portlet-name>Weather1Portlet</portlet-name>
  <portlet-class>examples.tutorial.weather1.Weather1Portlet</portlet-class>
</portlet>
```

2

Template overwiew

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.

```
@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`.

```
public class WeatherService {

  /** A cache for temperatures. */
  private final HashMap<String, String> cache = new HashMap<String, String>();

  public WeatherService() {
    System.out.println("aezfzef");
  }

  /**
   * Returns the temperature for the specifed location in celsius degrees.
   *
   * @param location the location
   * @return the temperature
   */
  public String getTemperature(String location) {
    String temperature = cache.get(location);
    if (temperature == null) {
      cache.put(location, temperature = retrieveTemperature(location));
    }
    return temperature;
  }

  private String getValue(String url, String xpath) throws Exception {
    XPathExpression expr = XPathFactory.
        newInstance().newXPath().compile(xpath);
    InputSource src = new InputSource(url);
    return expr.evaluate(src);
  }

  /**
   * Retrieve the temperature.
   *
   * @param location the location
   * @return the temperature
   */
```

```
  protected String retrieveTemperature(String location) {
    try {
      // First we get the location WOEID
      String woeidURL =
          "http://query.yahooapis.com/v1/public/yql" +
              "?q=select%20*%20from%20geo.places%20where%20text%3D%22" +
              URLEncoder.encode(location, "UTF-8") +
              "%22&format=xml";
      String woeid = getValue(woeidURL, "//*[local-name()='woeid']/text()");

      // Now get weather temperature
      String weatherURL =
          "http://weather.yahooapis.com/forecastrss?w=" +
              URLEncoder.encode(woeid, "UTF-8") +
              "&u=c";
      return getValue(weatherURL, "//*[local-name()='condition']/@temp");
    }
    catch (Exception e) {
      // Unavailable
      return "?";
    }
  }
}
```

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:

```
@Inject
WeatherService weatherService;
```

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

```
@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 init param defined in the deployment descriptor:

Example 3.1. Using Spring IOC in a servlet

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

The same can be achieved for a portlet of course:

Example 3.2. Using Spring IOC in a portlet

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

In the case of *Spring* injection, 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, however they are not covered in this tutorial.

4

Views

So far we 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.

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

The `@Route("/show/{location}")` annotation binds the the `index` controller method to the */show/** route. Since our our application package is annotated with the `@Route("/weather4")` annotation, an URL like */weather4/show/marseille* invokes this controller method with the *marseille* location.

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 changing view parameters when they are processed. Such 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.

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 Juzu, 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 this example, we use a form which contains the the `location` action parameters. In order to create an action url we use the same syntax than 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
  @Route("/add")
  public Response.View add(String location) {
    locations.add(location);
    return Weather_.index(location);
  }
```

The method is annotated with two annotations:

- `@Action` declares an action controller method
- `@Route("/add")` binds the action on the */weather5/add* route

The `location` parameter is not declared in the route because it can be invoked by a form and the `location` parameter is part of it.

The method returns a `Response.View` object. This object instructs Juzu to use the `index(String location)` controller view after the action. The `Weather_` controller compagnion provides a type safe way for generating the response: `Weather_.index(location)`. Our action just needs to return it, Juzu will take care of showing the corresponding view after invoking the action.

6

Type safe templating

We have seen previously how to render templates from a controller and how we can pass parameters for rendering a template. Templates use `${ }` expressions that often refers to parameters passed by the controller. For this purpose we used an `HashMap` for storing the various parameters used during rendering.

This syntax is a generic way and uses 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 example the `location` parameter is declared by the `#{param name=location/}` tag. During the Java compilation, Juzu leverage such parameter declarations to provide a more convenient way to render a template.

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

```
@View
public void index(String location) {
  index.
    with().
    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.

```
@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

Styling the application

Our application is almost complete, in this section we will work on the look and feel to make the application appealing. We will use the famous Twitter Bootstrap framework and show how it integrates well in Juzu applications.

7.1. The style

Bootstrap provides a solid fundation for building quickly attractive applications. Bootstrap is based on the dynamic stylesheet language Less and provides a very modular organization. We will perform trivial modifications to a subset of the Less files and integrate them in our application.

7.1.1. A la carte

We will then modify the `bootstrap.less` file to keep only what is necessary for our application:

Example 7.1. The necessary Bootstrap less files

```
/*!
 * Bootstrap v2.0.3
 *
 * Copyright 2012 Twitter, Inc
 * Licensed under the Apache License v2.0
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Designed and built with all the love in the world @twitter by @mdo and @fat
 */

// CSS Reset
@import "reset.less";

// Core variables and mixins
@import "variables.less"; // Modify this for custom colors, font-sizes, etc
@import "mixins.less";

// Grid system and page structure
@import "scaffolding.less";

// Base CSS
@import "type.less";
@import "forms.less";

// Components: common
@import "component-animations.less";

// Components: Buttons & Alerts
@import "buttons.less";
@import "button-groups.less";

// Components: Misc
@import "accordion.less";
```

This version of bootstrap.less is a trimmed down of the original files.

7.2. Plugins in action

Juzu can be extended with plugins, in this section we will use two of them

- The Less plugin compiles less files into css files
- The Asset plugin inject asset such as stylesheet or javascript in the application page

7.2.1. Less compilation

Juzu provides native support for the Less language via the Less plugin and the `@Less` annotation. It allows a set of less files to be transformed into the corresponding css files during the java compilation, achieving two important steps during the compilation phase:

- The less files are transformed into ready to use css files
- It ensures a maximum of safety: the Less parser will report any error in the source

Our first step is to create the `examples.tutorial.assets` package, we copy the Bootstrap Less files into this package and annotate the `examples.tutorial` package with the `@Less` annotation to trigger the compilation of the stylesheet in the `assets` package:

```
@Less(value = "bootstrap.less", minify = true)
package examples.tutorial;

import juzu.plugin.less.Less;
```

This annotation triggers the compilation of the `bootstrap.less` in the `assets` package, the `minified` parameter instruct Less to minify the resulting css.

7.2.2. Injecting CSS

Now that we have worked out the CSS details we need to make our stylesheet available in the application page. The *asset* plugin will achieve this result for us. This plugin provides declarative configuration for the various assets required by an application. It works both for stylesheets and javascript, in this section we use it for stylesheet:

Example 7.2. Injecting Bootstrap CSS in our application

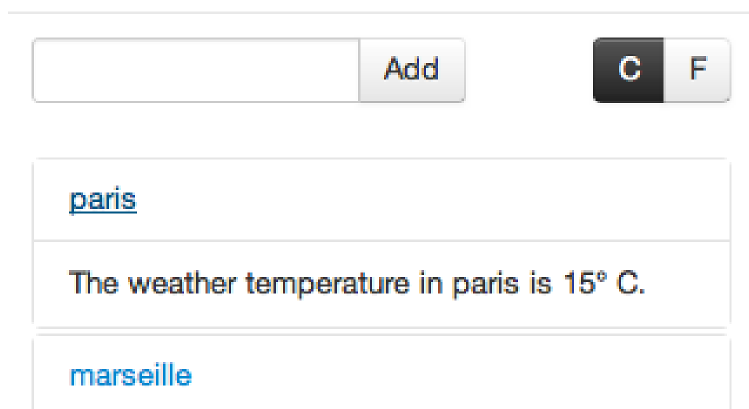
```
@Application
@Assets(stylesheets = @Stylesheet(
  src = "/examples/tutorial/assets/bootstrap.css",
  location = AssetLocation.CLASSPATH)
)
package examples.tutorial.weather7;
```

The usage is fairly straightforward with the `@Assets` and `Stylesheet` annotations. We configure the `location` parameter to be `CLASSPATH` because the Less plugin created it there.

7.2.3. Bringing CSS to life

After this step we need to modify our application template to use the various styles provided by Bootstrap:

Figure 7.1. The Bootstrapized application



We will not explain that in details, however we will study the important modifications:

7.2.3.1. Accordion

The Bootstrap provides the Collapse component. We will not use the entire Collapse component here but instead reuse the CSS rules to display the available cities:

```
<div class="accordion-group">
  <div class="accordion-heading"><a class="accordion-toggle" href="@{index(loca
  <div class="accordion-body">
    <div class="accordion-inner">The weather temperature in ${current} is ${ter
  </div>
</div>
```

7.2.3.2. Adding a city

Finally the form for adding is modified to reuse Bootstrap form support:

```
<form action="@{add()}" method="post">
  <fieldset>
    <div class="controls">
      <div class="input-append">
         <input class="span2" type="text" size="16"name="location" value=""/>
         <button type="submit" class="btn">Add</button>
      </div>
    </div>
  </fieldset>
</form>
```

8

Adding Ajax

Now that our application look great, we are going to add a final touch to our application and make it more dynamic.

In the previous section we introduced the accordion user interface component, but it was used in a static way. In this section we will make it dynamic and introduce the Ajax plugin.

The accordion component can be combined to the JQuery collapse plugin providing the capability to unfold an item to display the weather of the location. When the item is unfolded an ajax request is performed to the application to retrieve the markup that will be inserted.

8.1. Javascript to the rescue

The application will use several Javascript libraries:

- The JQuery library provides the fundations for building our application
- The Bootstrap accordion component provides scripts as a JQuery plugin
- Juzu provides Ajax helpers as a JQuery plugin
- A small custom script to setup the accordion plugin with the ajax plugin

8.1.1. Adding scripts

The Asset plugin was introduced in the previous section to handle the serving of the Bootstrap stylesheets. The `@Asset` annotation can be used also for embedding scripts, but it provides more control about the script, in particular a dependency mechanism to control the order of scripts that we will use: indeed JQuery plugins have to be loaded after the JQuery library is loaded.

We extend the `@Asset` annotation to add our scripts:

```
@Assets(
   scripts = {
      @Script(
         id = "jquery", ❶
         src = "jquery-1.7.1.min.js",
         location = AssetLocation.CLASSPATH),
      @Script(
         id = "transition",
         src = "bootstrap-transition.js", ❷
         location = AssetLocation.CLASSPATH,
         depends = "jquery"), ❶
      @Script(
         id = "collapse",
         src = "bootstrap-collapse.js",
         location = AssetLocation.CLASSPATH,
         depends = {
            "jquery", // 1
            "transition"}), ❷
      @Script(
         src = "weather.js",
         location = AssetLocation.CLASSPATH,
         depends = {
            "jquery",  ❶
            "collapse"}) ❸
   },
   stylesheets = @Stylesheet(src = "/examples/tutorial/assets/bootstrap.css",
)
package examples.tutorial.weather8;
```

❶ All other scripts than JQuery depends on JQuery

❷ The *collapse* JQuery plugin depends on the *transition* script

❸ Our *weather* script depends also on the JQuery *collapse* plugin

The declaration is straightforward, any `@Script` annotation may configure

- an optional `id` used for creating dependencies between scripts
- a mandatory `src` for the script name
- an optional `location` for resolving the script
- an optional `depends` that declares the ids a script depends on

8.1.2. The collapse plugin

The Bootstrap collapse plugin allows the user to unfold a city to display its weather, you can see how it works in the Bootstrap manual. In our case we modify the `index.gtmpl` to use it, in particular the `accordion-group` part:

```
<div class="accordion-group">
  <div class="accordion-heading">
    <a class="accordion-toggle" href="#${current}" data-parent="${id}" data-to
  </div>
  <% def expanded = i != index ? 'in' : ''; %>
  <div id="${current}" class="accordion-body collapse ${expanded}">
    <div class="accordion-inner">
    </div>
  </div>
</div>
```

There are two noticeable points to explain:

- The `accordion-inner` is empty now, the reason is that the weather will be loaded using JQuery ajax capabilities
- The `div.collapse` element id is set to the current item location, this value will be reused during an ajax interaction, we will see more about this later.

8.2. Bridge the gap with Ajax

Now our application loads a set of dynamic tabs managed by JQuery and the collapse plugin, it's time to develop the ajax part: our goal is to load a markup fragment to insert in an accordion item when it is unfolded. We will develop a bit of client side Javascript and a resource controller on the Weather controller.

8.2.1. Resource controller

We need a controller method to server the markup of the weather of a particular city. Until now we have studied view and action controllers, for this use case we will use a new type of controller : the *resource* controller.

Resource controllers are pretty much like a view controllers except that they must produce the entire response sent to the client and that is precisely what we want to achieve on the server side.

Our application requires a single resource controller method that we will call `getFragment`, it will render a markup fragment for a specific city location:

```
  @Inject
  @Path("fragment.gtmpl")
  examples.tutorial.weather8.templates.fragment fragment;

  @Ajax
  @Resource
  @Route("/fragment")
  public void getFragment(String location) {
    fragment.
      with().
      location(location).
      temperature(weatherService.getTemperature(location)).
      render();
  }
```

The fragment template is very simple and only renders the portion of the screen that will be updated by the client side javascript code when an item is unfolded:

```
#{param name=location/}
#{param name=temperature/}
<p>The weather temperature in ${location} is ${temperature} degrees C</p>
```

8.2.2. JQuery

The last part of our application is the javascript part that will react on the collapse plugin component to load the markup from the `getFragment` controller when the item is unfolded. We create the javascript file `weather.js` in the `examples.tutorial.weather8.assets`:

```
$(function () {

  // Setup an event listener on the .collapse element when it is shown
  $('.collapse').on('show', function () {

    // Get the location attribute from the dom
    var location = $(this).attr("id");

    // Update the .accordion-inner fragment
    $(this).
      closest(".accordion-group").
      find(".accordion-inner").
      each(function () {

      // Load the fragment from the resource controller
      $(this).jzLoad(
        "Weather.getFragment()",
        {"location":location});

    });
  });

  // Setup the collapse component
  $(".collapse").collapse();
});
```

The following construct is important because it ensures that the code inside the function will be executed when JQuery is loaded:

```
$(function() {
  //
});
```

If you take time to read the collapse plugin component documentation you will see that there are two important things to do for integrating it in our application:

- We must setup an event listener on `.collapse` elements to react to the `shown` event
- The collapase component is setup with the `$(".collapse").collapse()` code

The important code is inside the `show` event listener:

- The location to display is extracted from the element `id` on the `div.collapse` component
- We find the appropriate element to update with the `$(this).closest(".accordion-group").find(".accordion-inner")` selectors
- On the `.accordion-inner` element we invoke the resource controller with the `jzLoad(...)` function

The `jzLoad` function is a JQuery plugin provided by the Juzu ajax plugin. It allows to invoke a controller method using ajax and cares about propagating the call to the resource controller method. It replaces the standard JQuery `load` function and accepts the same argument. However the url part is replaced by a controller method, `Weather.getFragment()` in our case.

The Juzu ajax plugin takes care of finding the right URL for invoking the controller method. It is designed to work in a standalone javascript file without requiring `<script>` tags in the page and works even when multiples instances of the portlets are on the same page.

When the ajax plugin operates on `@Ajax` controller method, it wraps the markup with a DOM structure that contains the URL for the `@Ajax` method of the application:

```
<div class="jz">
  <div data-method-id="Weather.getFragment()" data-url="http://localhost:8080/
  ...
</div>
```

The `jzLoad` function is invoked on a portion of the DOM that is wrapped by the `div.jz` element and this function will simply locate the correct URL using the `data-method-id` attribute. This makes the same script work with different portlets on the same page.

Note also that in our code we never used any JQuery selector containing id in order to allow several instances of the same application to work without conflicts.

9

Testing our application

The last chapter of our tutorial will teach you how to test a Juzu application. Juzu applications can be tested using existing tools, we will use in this chapter the following tools:

- JUnit 4
- Arquillian : a framework for managing web containers
- ShrinkWrap: Arquillian's little brother for creating Java archives easily
- Selenium WebDriver : a simple API for simulating browser behavior

For making testing easy, Juzu provides a Maven dependencies containing all the required dependencies for testing an application:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-bom-arquillian</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-bom-arquillian-tomcat7</artifactId>
  <scope>test</scope>
</dependency>
```

The *juzu-bom-arquillian* and *juzu-bom-arquillian-tomcat7* provides setup for Arquillian and Selenium for Tomcat7 based testing.

9.1. Setting up the test

Let's start by setting up our test class with Arquillian, the goal is to run the Weather application during the test. We will rely on the Tomcat servlet container for running our application and on the Arquillian framework for starting and stopping Tomcat. Arquillian provides a JUnit runner for managing a web container during a unit test:

Example 9.1. Using the Arquillian runner

```
@RunWith(Arquillian.class)
public class WeatherTestCase {
}
```

Arquillian supports also the TestNG framework

This only setup Tomcat during the test, we need to deploy the Weather application and for this we use Arquillian `@Deployment` annotation and we return a ShrinkWrap `WebArchive` object that will be deployed in Tomcat by Arquillian. `WebArchive` are easy to build programmatically, however we will use an helper provided by Juzu to build the base archive:

Example 9.2. Application deployment

```
@Deployment
public static WebArchive deployment() {
  WebArchive war = Helper.createBaseServletDeployment(); ❶
  war.addAsWebInfResource(new File("src/test/resources/spring.xml")); ❷
  war.addPackages(true, "examples.tutorial"); ❸
  return war;
}
```

❶ Create the base servlet deployment

❷ Add the spring.xml descriptor

❸ Add the examples.tutorial package

For testing our application we will use Selenium WebDriver managed by Arquillian. Arquillian can inject WebDriver thanks to the *Drone* extension and it is quite easy to achieve. We also need the base URL of the Weather application after it is deployed:

Example 9.3. Arquillian injection

```
@Drone
WebDriver driver;

@ArquillianResource
URL deploymentURL;
```

The last step of the setup is a little helper method for creating application URL for our applications *weather1*, *weather2*, ...

Example 9.4. Creating URL for an application

```
public URL getApplicationURL(String application) {
  try {
    return deploymentURL.toURI().resolve(application).toURL();
  }
  catch (Exception e) {
    AssertionFailedError afe = new AssertionFailedError();
    afe.initCause(e);
    throw afe;
  }
}
```

This method simply generates an URL based on the application name, for example `getApplicationURL("weather1")` returns the URL for the *weather1* application.

9.2. Testing the app

Now that our test class is done we can write a few tests for the application:

Example 9.5. Creating URL for an application

```
@Test
@RunAsClient
public void testWeather1() throws Exception {
  URL url = getApplicationURL("weather1"); ❶
  driver.get(url.toString());
  WebElement body = driver.findElement(By.tagName("body"));
  assertTrue(body.getText().contains("The weather application")); ❷
}
```

❶ Retrieve the application

❷ Check markup is correct

10

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.