Table of Contents

List of Examples

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.

1

Build, deploy and run

We will see in this chapter how to build and deploy a Juzu application.

1.1. Build

Building a Juzu application is usually done in two steps

- Compile the application to its binary representation

- Package the application as a war file

Compiling an application requires a few jars to be present on the compilation classpath:

- The Juzu core jar for the Juzu API

- The JSR-330 jar for the @Inject API

- Any Juzu extension jar such as plugins or additinal template engines

After compilation, classes need to be packaged as a web application archive (*war*) and then deployed in a server. We will show several ways to package a Juzu application.

> At the moment Juzu focuses on Maven because it is built with Maven, however that does not mean that Juzu is coupled to Maven, in the future we will provide additional examples or quickstart for alternative build systems.

1.1.1. With Maven

Juzu libraries are deployed in the Maven Central repository, compiling an application with require a few dependencies to find the correct jars.

1.1.1.1. Using the Juzu Maven *bom*

The *bom* is a Juzu artifact that serves the purpose of building and packaging an application:

1. provide a set of dependencies that will be sufficient for compiling the application using its Maven transitive dependencies

2. provide a predefined assembly descriptor that creates a war file containing the application classes, resources and libraries

To achieve the first step, we simply declare the following dependency in a Maven artifact:

```xml
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-bom-core</artifactId>
  <version>0.7.0-beta15</version>
</dependency>
```

Assembling the application requires more XML but is very straightforward:

```xml
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.juzu</groupId>
      <artifactId>juzu-bom-core</artifactId>
      <version>0.7.0-beta15</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <goals>
        <goal>single</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <descriptorRefs>
          <descriptorRef>tomcat-guice</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The *assembly* plugin takes care of packaging the application:

1. The plugin dependency declares on the *juzu-bom-core* artifact because it contains the predefined descriptors such as the `tomcat-guice` descriptor

2. The goal *single* of the assembly plugin is executed during the *package* phase

3. The predefined descriptor *tomcat-guice* packages the application for the Tomcat server and with the Guice framework

   1. Any dependency on the application is packaged in *WEB-INF/lib*

   2. The application classes are copied in *WEB-INF/classes*

   3. The web application *src/main/webapp* files are copied to the root of the archive

   4. Specific deployment descriptors may be copied in the war file depending on the

predefined descriptor

In this example we used the *tomcat-guice* predefined descriptor. The bom relies on the Maven Assembly plugin, Juzu provides the predefined assembly descriptors that makes easy to package a Juzu application:

Table 1.1. The predefined descriptors

	tomcat	gatein	liferay
all	tomcat	gatein	liferay
guice	tomcat-guice	gatein-guice	liferay-guice
spring	tomcat-spring	gatein-spring	liferay-spring
cdi	tomcat-cdi	gatein-cdi	liferay-cdi

> The predefined assembly descriptor does a similar job to the Maven *war* packaging but with more flexibility. To achieve the same result, the usage of a war packaging with the overlay feature.

1.1.1.2. Juzu archetype

The following produces a base Juzu application for Tomcat with the Guice injection container:

Example 1.1. Generating an application for the Tomcat server and the Guice injection contai

```
mvn archetype:generate \
    -DarchetypeGroupId=org.juzu \
    -DarchetypeArtifactId=juzu-archetype \
    -DarchetypeVersion=0.7.0-beta15 \
    -DgroupId=org.example \
    -DartifactId=myapp \
    -DpackageName=org.example.myapp \
    -Dversion=1.0.0-SNAPSHOT
```

The generated application is a quickstart ready to can be customzed for developing more complex applications. The archetype uses the packager described in the previous section.

It is possible to generate the application for a different server and injection container:

Example 1.2. Generating an application for the Tomcat server and the Spring injection contai

```
mvn archetype:generate \
   -DarchetypeGroupId=org.juzu \
   -DarchetypeArtifactId=juzu-archetype \
   -DarchetypeVersion=0.7.0-beta15 \
   -DgroupId=org.example \
   -DartifactId=myapp \
   -DpackageName=org.example.myapp \
   -Dversion=1.0.0-SNAPSHOT \
   -DjuzuServer=tomcat \
   -DjuzuInject=spring
```

1.1.2. Using a prepackaged application

The Juzu distribution contains the Booking and Tutorial applications for GateIn and Liferay servers. They can be used as basis to create applications.

1.1.3. Using an IDE

Juzu uses Annotation Processing Tool to perform many tasks at compilation time. APT is a standard extension of a Java compiler. All Java IDE (Eclipse, Intellij and Netbeans) provide good support for APT, we will show in the section how to configure and uses APT within those IDEs.

IDEs provide also Maven support, we will focus in this section on using APT without the Maven support. Indeed the APT support may work differently when using Maven in your project, the Maven and APT support within IDEs has a dedicated section.

1.1.3.1. Intellij support

todo

1.1.3.2. Eclipse support

todo

1.1.3.3. Netbeans support

todo

1.2. Deploy

At the moment the supported (i.e tested) portal servers are

- Tomcat 6.x and 7.x

- GateIn 3.2 / 3.3 / 3.4 and 3.5

- Liferay 6.1

Other server may work but we are not aware of that as it was not tested in other environments.

1.2.1. Tomcat

No specific deployment instruction.

1.2.2. GateIn

1.2.2.1. GateIn on Tomcat 6/7

No specific deployment instruction.

1.2.2.2. GateIn on JBoss AS 7

GateIn on JBoss AS7 requires a little modification to do:

Open the file *modules/javax/api/main/module.xml* and add ***<path name="javax/annotation/processing"/>*** among the *paths* declaration:

```
<module xmlns="urn:jboss:module:1.1" name="javax.api">
  <dependencies>
    <system export="true">
      <paths>
        <path name="javax/annotation/processing"/>
        ...
      </paths>
    </system>
  </dependencies>
</module>
```

This configuration exposes the `javax.annotation.processing` package to the classes seen by Juzu.

1.2.3. Liferay

Liferay has been tested extensively with the Tomcat version, no specific deployment instruction is required.

1.3. Run

1.3.1. Run modes

Juzu defines three modes for running an application, called *run modes*:

- Production (*prod*): error reporting is minimal

- Development (*dev*): provides verbose error reporting

- Live (*live*): allow to develop the application live with verbose error reporting

1.3.2. How to choose the right run mode

- When you are running an application use the *prod* run mode

- When you are developing an application with a build system use the *dev* run mode

- When you are live developing an application use the *live* run mode

1.3.3. Configuring the run mode

Run mode is configured via servlet context parameters, by default the *prod* mode is enabled:

- *juzu.run_mode* : possible values *prod*, *dev* or *live*

- *juzu.src_path* : the source path of the source to compile for the live mode

Example 1.3. Configuring the dev mode in web.xml

```
<context-param>
  <param-name>juzu.run_mode</param-name>
  <param-value>dev</param-value>
</context-param>
```

2

Phases

Request life cycle is the most important concept to get right when developing a web application, whether it is a Juzu application or not. Juzu maps a request to a phase, in this chapter we will explain the phase concept and its importance.

2.1. Phases

Juzu request life cycle is composed of four phases, we will explain three of them in this chapter, the last one will be explained in another chapter of this guide.

- The view phase : invoke the application to produce markup output aggregated within a page

- The action phase : invoke the application to process an action

- The resource phase : invoke the application to produce any kind of output as a full response (i.e not in a page)

During the execution of a phase, parameters are provided by Juzu to execute the phase. Those parameters are set by the application itself, for instance when it creates a link. The scope of the parameters (i.e when the validity of the parameters) depends on the phase.

2.1.1. View phase

The view phase invokes the application for the purpose of creating markup. This phase is indempotent, it means that repeated invocation of the same phase with the same parameters should produce the same markup (supposing than the application does not depend on some other state, like a database that could change over time).

View parameters are associated with the current URL, it means that they are somehow persistent. For instance you interact with an application to change its view parameters on each request and then you interact with another application on the same page: the view parameters will remain the same accross the invocation of the view phase of the application when the second application is used.

2.1.2. Action phase

The action phase invokes the application for processing an action. During the invocation, action parameters are provided and their validity is limited to the current action phase being executed, after they will not be anymore available.

The action phase is not idempotent, invoking several times an action phase could have side effects such as inserting several times the same data in a database.

Juzu does not expect markup returned during this phase, however it provides the opportunity to configure the view parameters of the next view phase.

2.1.3. Resource phase

The resource phase allows the application to produce a web resource such as an image or a full page. When this phase is invoked, a set of resources parameters are provided in the URL producing the resource.

2.2. Interactions

Now that we have an overview of the phase, it is time to connect them and explain the interactions between the phases.

Figure 2.1. Interaction between phases
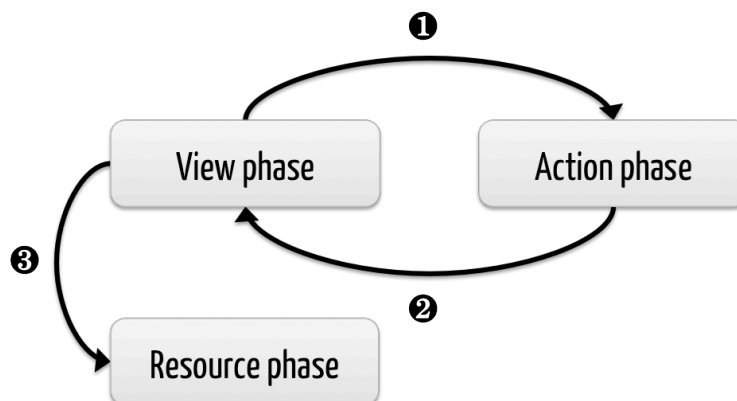
1. An action phase is invoked by an URL produced during a view phase, this URL contains the action parameters

2. After an action phase a view phase is executed and the view parameters are updated

3. A resource phase is invoked by anURL produced during a view phase, this URL contains the resource parameters

2.3. Mapping onto HTTP

As said before, phases and interactions have a natural mapping with the HTTP protocol. It is worthy to explain it because it will help you to understand fully the interations managed by Juzu.

2.3.1. View phase

View phases are mapped on *GET* requests:

- The view phase is idempotent like GET

- View parameters are identified to query parameters

- The response returned by a GET request should remain identical for the same parameters

During a view phase, the application produces URL which can invoke any application phase.

Figure 2.2. View phase

GET http://...?color=red

View phase

↓

color=red
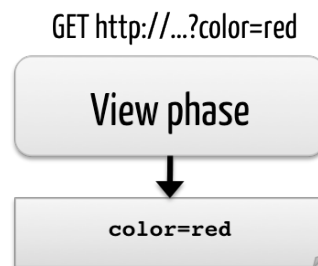
In this example the view phase produce markup parameterized by the `color` parameter having the *red* value.

2.3.2. Action phase

Action phase are created from view phase by processing a link that was found in the markup response. The action phase is mapped on *POST* requests:

- Both action phases and POST request are not idempotent

- Action parameters are identified to form parameters

- Action phase and POST requests should not be invoked more than one time

Figure 2.3. Action phase

```
        color=red
<form action="…"
method="post">...</
form>
```

POST http://...

Action phase

Now let's update our example and suppose that the application returns markup with a form that invokes an action phase. When the user submits the form it triggers the action phase, which in returns updates the `color` view parameter of the next view phase to the value *blue*.

Figure 2.4. View phase after action phase
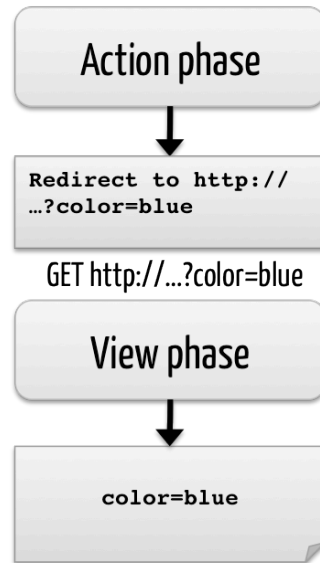
The HTTP redirection will update the browser to show the next view phase with the expected view parameters.

During the action phase, the application configures the parameters of the next view phase. When the invocation of the phase is over, the server redirects the browser (with an HTTP temporary redirection) to the next view phase URL. This URL contains the view parameters. This mechanism is well known as *Redirect After Post* pattern and is often used to ensure that a POST request is not triggered several times when the refresh button of the browser is used.

2.3.3. Resource phase

Resource phases are trivially mapped on *GET* request pretty much like a view phase. The main difference is that the resource phase is responsible for managing the entire response instead of just a fragment of the response.

3

Controllers

Controllers play an essential role in a Juzu application: they contain the code executed when Juzu processes a request, this chapter provides an in depth study of Juzu controllers.

3.1. Overview

Juzu controllers are simply annotated methods of the application, here is the most basic controller declaration:

```java
public class Controller {
  @View public Response.Content index() {
    return Response.render("hello world");
  }
}
```

The annotation `@juzu.View` declares a *view* controller, the name `index` has a special meaning as it will be used when no other controller is specifed in a Juzu request.

Controller methods can declare parameters for receiving request parameters:

```java
public class Controller {
  @View public Response.Content index(String person) {
    return Response.render("Hello " + person == null ? "world" : person);
  }
}
```

Like previously, the `index` controller returns the *hello world* value when it is called the first time. When the controller is called with the `person` parameter it returns the hello string personalized with the corresponding parameter value: Juzu use the declared method parameter name to match against the request parameters, in our case the `person` request parameter.

Any controller class (any class containing at least one controller method) generates a *companion* class during the compilation of the project. Such companion class extends the original controller class to provider companion methods for the controller method. The companion class has the same name than the original class appended with the _ character:

```java
public class Controller_ {
  public static Dispatch index() { /* Generated code */ }
  public static Dispatch index(String person) { /* Generated code */ }
}
```

Each `index` methods generated a corresponding `index` method companion. When any `index` method is invoked it returns an `juzu.Dispatch` object that generates the URL dispatching to the corresponding phase when the `toString()` method is invoked. When parameters are provided they will be encoded in the generated URL.

```java
@View public Response.Content index() {
  return Response.render("Hello word. <a href='" + Controller_.index("Juzu") +
}
```

URL companion methods have the name of the originating method appended with the *URL* suffix. The method parameter types are the same.

3.2. Request routing

During a request, Juzu routes the request to the correct controller method. Previously we have seen that any unmatched view phase request will be handled by the `index` controller method.

In this section we cover the binding of a controller method to a specific request. This binding is not the same whether you are writing an application that deploys as a servlet or as a portlet.

The main difference between the two environements are the request and responses: servlets interacts with the http protocol whereas the portlets interacts with the portal (which can turns into the WSRP prococol in the case of a remote portlet).

In practice the main difference between servlet and portlet is the routing of the request: with a servlet the controller methods needs to be bound to route.

3.2.1. Http request routing

When Juzu handles an http request it routes this request to a controller based on the request path. Request routing is based on a set of route declarations, each declaration binds a route to a controller method.

3.2.1.1. Default controller

Before discussing routing configuration, we shall remind the default controller method: the `index` view controller method of the default controller will handle any unmatched request:

```java
@View
public void index() {
  // Handle unmatched request
}
```

When the application has a single controller class, the default controller is this controller. When there are more than one controller, there is an ambiguity. In this situation the default controller

should be specified in the `@Application` annotation:

```
@Application(defaultController = Controller.class)
```

3.2.1.2. Declaring a route

The `@Route` annotation declares the route for a controller with a `path` and an optional `priority`:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Route {

  /**
   * The route path.
   *
   * @return the route path
   */
  String value();

  /**
   * The route priority.
   *
   * @return the route priority
   */
  int priority() default 0;

}
```

Controller method should declare an `@Route` annotation, in practice with http the annotations `@View`, `@Action` and `@Resource` are associated with an `@Route` annotation.

Example 3.1. Declaring a controller route

```
@View @Route("/show")
public void show() {
  ...
}
```

The request *show* will be dispatched to the `show()` method.

3.2.1.3. Route parameters

Route can declare parameters:

Example 3.2. Route parameters

```
@View @Route("/show/{id}")
public void show(String id) {
  ...
}
```

In this example the route parameter `id` will match the controller method parameter `id` and a

request like */show/123* will invoke the `show(String id)` method with the *123* value.

3.2.1.4. Route parameter pattern matching

Optionally, route parameters can match regular expression. This can be achieved with the `@Param` annotation:

Example 3.3. Route parameter matching

```
@View @Route("/show/{id}")
public void show(@Param(pattern="[0-9]+") String id) {
  ...
}
```

3.2.1.5. Route overloading

The same route can bound to different phases, the dispatch behavior depends on the http method:

- in a *GET* method the phases priority are *view*, *action*, *resource*

- in a *POST* method the phases priority are *action*, *view*, *resource*

Example 3.4. Route parameter matching

```
@View @Route("/show")
public void showWithView() {
  ...
}

@Action @Route("/show")
public void showWithAction() {
  ...
}
```

With those rules:

- A *GET* request on the */show* path will invoke the `showWithAction()` method

- A *POST* request on the */show* path will invoke the `showWithView()` method

3.2.1.6. Route priorities

When several routes match the same request, the router will use the first route found. The `priority` parameter of the `@Route` annotation can be used to increase the priority of a route. This can be useful, specially when a route contains a parameter that could match another route instead.

Example 3.5. Route priority

```java
@View @Route("/show/status", priority = 1)
public void showStatus() {
  ...
}
@View @Route("/show/{name}")
public void show(String name) {
  ...
}
```

In the example, the `showStatus()` controller will be invoked when the route */show/status* is requested. Without this higher priority, the `show(String name)` controller might be invoked instead. When no priority is specified, the default priority is **0**.

3.2.1.7. Redirect after post

As explained in the phases chapter, an action never produces markup, instead an action phase is followed by a view phase that will return a markup response. Juzu handles this interaction with an http redirection to the next view phase via the redirect after post pattern.

This behavior is good for the user because the browser will be updated with an URL of the view phase that is bookmarkable and safely refreshable (i.e the user an refresh the page safely).

However Juzu does not enforce this behavior and it can be changed to have the view phase immediatly invoked after the action phase.

```java
@Action
@Route("/process")
public Response.View process() {
  return Controller_.index().withNo(PropertyType.REDIRECT_AFTER_ACTION);
}

@juzu.View
@Route("/show")
public void show() {
  //
}
```

3.2.2. Portlet request routing

Unlike the http protocol, the portlet request routing does not require the `@Route` annotation because portlet requests are managed by the portal and have no concept of path mapping.

To achieve request routing, the portlet uses a special portlet request parameter named *juzu.op* . This parameter determines which controller should be called during a phase. When the *juzu.op* parameter is not present, Juzu will look for the `index` view controller.

3.3. Controller phases

There are several kinds of controllers bound to a request phase studied in the ???:

- View controllers annoted with the `@juzu.View` annotation

- Action controllers annotated with the `@juzu.Action` annotation

- Resource controllers annotated with the `@juzu.Resource` annotation

- Event controllers annotated with the `@juzu.Event` annotation (*not yet implemented*)

3.3.1. View controllers

A view controller method produces aggregated markup for the application, the invocation of the method should produce markup that will be aggregated in larger page, therefore it should not care about the overall HTML structure.

View parameters describe the current parameters of the view, they are often used for navigation purpose in the application. Juzu supports simple data types such as string and structured data types modelled by Java objects.

- Simple data types can be the following types `String`, `List<String>` and `String[]`. Later this will be expanded to more simple types such as number, etc..

- Structured data types : todo

View controller method should return a `juzu.Response` object that is the content produced by the method. To be more precise it should return a `Response.Body` or `Response.Content` object (the latter being a subclass of the former) that contains everything Juzu needs to display the application.

`Response.Content` is a base class for content, it defines the `send` method. Juzu invokes this method when it needs to render the response produced by the view method. The invocation of the `send` method will be performed after the view method is invoked.

```
Could not locate the JavaCodeLink[juzu.Response.Content#send(juzu.io.Stream)] :
```

During the view phase a controller can generate URLs to other phases (except the event phase) by using controller companion methods. Companion methods returns a `juzu.Dispatch` object to represent the URL. The final URL is returned by the `toString()` method of the dispatch object.

3.3.2. Action controllers

Action controller are executed during the action phase of a Juzu application. Usually action methods perform two tasks

- implement the logic of the application processing, for instance inserting an entity in the database

- configure the next view phase: setting the next view controller to display and configuring its view parameters of the method when they exist

In the following example, the controller method `createUser` creates a user and returns a `Response.View` object that will tell Juzu to use the `showUser` view controller during the next

view phase:

```
@Action
public Response.View addUser(String userName, String password) {
  orgService.createUser(userName, password);
  return Controller_.showUser(userName);
}
```

`showUser` is a companion *view* method that creates a `Response.View` object configured with the controller and arguments to use. Like url companion methods, view companion methods are generated during the compilation of the project by Juzu.

3.3.3. Resource controllers

Resource controllers are similar to view controllers, however the resource has full control over the target page. It means that a resource controller must produce the entire resource and it can also chose the mime type returned. Resource controllers have several use cases:

- Implement ajax resource serving

- Produce an application resource, such as an image, a script, etc...

3.3.4. Event controllers

not yet implemented

3.4. Controller classes

Controller methods belongs to Java classes known as controller classes. Controller classes are ordinary java classes, any class can be turned into a controller by declaring a controller method. Controller classes are registered in the IOC container of the Juzu application, we will study later the benefits.

3.4.1. Controller life cycle

We will study in this section the complete life cycle of a controller object. Juzu relies on the IOC container for managing the life cycle of controller objects, based on the `@javax.inject.Inject` annotation. If the controller desires, it can receive life cycle callbacks thanks to the `@javax.annotation.PostConstruct` and `@javax.annotation.PreDestroy` annotations.

Let's have a look at the complete life cycle of a controller object during a Juzu request:

Figure 3.1. Life cycle of a controller object
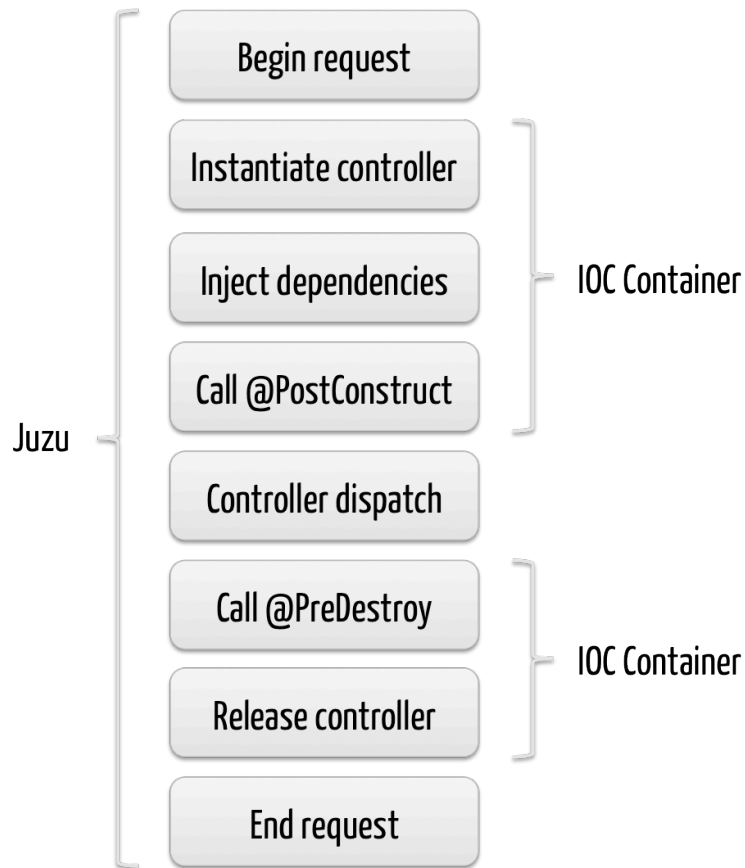
1. Juzu begins the request, it will need an controller instance for the request and asks the IOC container an instance

2. The IOC container creates a fully operational controller instance in several stesp

   1. It gets a controller object instance either by creating a new instance by using the default constructor or the constructor annotated with `@Inject`

   2. It injects the controller declared dependencies by the `@Inject` annotation

   3. It invokes any method annotated with `@PostConstruct`

3. Juzu obtains a valid controller instance and dispatches that method on the controller

4. After the invocation, Juzu releases the controller instance and delegates it to the IOC container again

   1. It invokes any method annotated with `@PreDestroy`

   2. It makes the instance available to the garbage collector

5. Juzu ends the request and use the `Response` objet returned by the controller method

3.4.2. Controller dispatch

The bare minimum Juzu will do when dispatching to a controller method is to invoke this method with the proper arguments and use the optionally returned object as a response for the request.

When the controller wants to deal with the dispatch in a generic manner (i.e detyped), it can implement the `juzu.request.RequestLifeCycle` interface that allows to:

- Be aware of the request life cycle around (i.e before and after) the controller method dispatch

- Control the response for the current request

Example 3.6. The RequestLifeCycle interface

```java
public interface RequestLifeCycle {

  void beginRequest(RequestContext context);

  void endRequest(RequestContext context);

}
```

- The `beginRequest` method is invoked before the controller method invocation and the `endRequest` is invoked after the controller method invocation.

- The `RequestContext` object provides a read/write access to a `juzu.Response` object that is set with the response returned by the controller method invocation. The controller method can declare no return type and instead set the response directly with the `RequestContext#setResponse(juzu.Response)` method.

- When the controller method invocation throws an exception, the `endRequest` method will be invoked with a `juzu.Response.Error` response set on the `RequestContext`.

- The `beginRequest` invocation can optionally set of response on the `RequestContext`, when it does the dispatch will stop here and the provided response will be the request response.

- The `endRequest` invocation can optionally set a response on the `RequestContext`, when it does this overrides the previous response provided during the dispatch to the controller method.

4

Responses

Each request produces a response object: a subclass of the `juzu.Response` class.

Response objects are returned by method processing phases, the class of the object determines the kind of response sent to the client. A response object may carry additional objects such as assets (css or script).

Response object are created thanks to the static factory methods of the `juzu.Response` class. The `Response` class is abstract and it has several subclasses that form a possible hierarchy of response adapted to the phase being processed.

4.1. Content responses

A *content* response is a markup or binary data, it can be created with the `ok` static method:

```java
public static Response.Content<Stream.Char> ok(CharSequence content) { ... }
```

It can be used during a *view* or *resource* phase to return markup:

```java
@View
public Response.Content index() {
  return Response.ok("Hello World");
}
```

4.2. Render response

A *render* response extends a *content* response, it specializes it for aggregated markup, i.e a response where the application manages only one portion of the full page such as a portal:

```java
@View
public Response.Content index() {
  return Response.ok("Hello World").withTitle("The Hello");
}
```

4.3. View response

View response is returned after the *action* phase to configure the next *view* phase. Usually view responses are not created directly using static factory methods, instead they are created using controller companion static methods, this will be explained in the controller chapter.

4.4. Redirect response

Redirect responses are returned during an *action* phase to redirect the user agent to an URL, its usage is simple:

```
@Action
public Response.Redirect process() {
  return Response.redirect("http://www.host.com/");
}
```

4.5. Error response

Error response can be returned during any phase to signal an application error. This kind of response is different from a content response with a 5xx status code, however it can be turned into a 5xx response to the client.

```
@View
public Response index() {
  try {
    ...
  } catch(IOException e) {
    return Response.error(e);
  }
}
```

An error response can also be generated by the controller method by declaring the exception in its throw clause, so the previous example is equivalent to:

```
@View
public Response index() throws IOException {
  ...
}
```

5

Bridges

The bridge is the runtime in which Juzu executes, until now Juzu provides two bridges:

- The servlet bridge executes a Juzu application in a servlet container like Tomcat

- The portlet bridge executes a Juzu application in a portlet container inside a portal

5.1. Servlet bridge

The servlet bridge exposes a Juzu application as a servlet in a Servlet Container.

5.1.1. Juzu servlet

The first step for using the servlet bridge is to configure the `juzu.bridge.servlet.JuzuServlet` servlet for the application. There is a one to one mapping between a Juzu application and a Juzu servlet. Therefore if you project contains several applications, you should configure a Juzu servlet for each.

5.1.2. Servlet configuration

Declaring a Juzu servlet is done in the *web.xml* file of the web application:

Example 5.1. The Juzu servlet configuration

```
<servlet>
  <servlet-name>JuzuServlet</servlet-name>
  <servlet-class>juzu.bridge.servlet.JuzuServlet</servlet-class>
  <init-param>
    <param-name>juzu.app_name&gt;</param-name>
    <param-value>my.application</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>JuzuServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

The *juzu.app_name* init parameter tells Juzu the package of the application to use. The servlet is bound on the / pattern as the default servlet of the web application.

In case of several applications, each can be configured with a *path mapping* in addition of the default servlet:

Example 5.2. Path mapping configuration

```
<servlet-mapping>
  <servlet-name>JuzuServlet</servlet-name>
  <url-pattern>/myapplication/*</url-pattern>
</servlet-mapping>
```

Any other kind of *url-pattern* than the default servlet (/) or path mapping is not supported and will raise an error during startup

5.2. Portlet bridge

The portlet bridge exposes a Juzu application as a portlet in a Portlet Container.

5.2.1. Juzu portlet

The first step for using the portlet bridge is to configure the `juzu.bridge.portlet.JuzuPortlet` portlet for the application. There is a one to one mapping between a Juzu application and a Juzu portlet. Therefore if you project contains several applications, you should configure a Juzu portlet for each.

5.2.2. Portlet configuration

Declaring a Juzu portlet is done in the *portlet.xml* file of the portlet application:

Example 5.3. The Juzu portlet configuration

```
<portlet>
  <portlet-name>JuzuPortlet</portlet-name>
  <display-name xml:lang="EN">Juzu Portlet Application</display-name>
  <portlet-class>juzu.bridge.portlet.PortletBridge</portlet-class>
  <init-param>
    <param-name>juzu.app_name</param-name>
    <param-value>my.application</param-value>
  </init-param>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <portlet-info>
    <title>Portlet Application</title>
  </portlet-info>
</portlet>
```

The *juzu.app_name* init parameter tells Juzu the package of the application to use.

6

Inversion of Control

Juzu provides native support for Injection of Control (known as *IOC*) and relies on the specification JSR 330 (known as *@Inject*).

Although the JSR-330 is quite small it provides the necessary ground for building Juzu applications. Juzu relies on the injection container for wiring the entire Juzu runtime (controllers, templates, plugins, etc...).

We will explain how Juzu uses IOC for its runtime, we suppose the reader is familliar with IOC and with the `@Inject` specification, in particular the notion of injection, scope and qualifier should be familliar.

6.1. Containers

At the moment Juzu supports three containers implementing the JSR 330:

- Spring Core 3

- Context and Dependency Injection also know as *CDI* implemented by the Weld project

- Google Guice 3

> CDI is a specification that extends the *@Inject* specification: CDI provides more features than @Inject, however this specification is only implemented by Weld. Nevertheless if your choice is to use CDI you will be leverage its specific features in your Juzu application

Juzu can run with any of those implementation and leaves you the choice of the IOC implementation you want to use. The container to selection is done via the servlet context parameter *juzu.inject*:

- *guice* for Google Guice

- *spring* for Spring

- *weld* for JBoss Weld

Example 6.1. Configuring the Spring container in web.xml

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

When no IOC container is specified the Guice container will be used

6.2. Inversion Of Control

6.2.1. Beans

Beans are simply object managed by the IOC container, any bean can be injected other beans:

```
@java.inject.Inject
Service service;
```

6.2.2. Scopes

Scopes define how a instances of a bean are managed by the IOC container: for a given bean, shall it be instantiated only one time and shared or shall it instantiated everty time it is required ? that's the kind of question that scope answers.

Juzu provides 4 scopes to use within your application:

- `@javax.inject.Singleton` scope: a single bean instance is the same for the whole application

- `@juzu.RequestScoped` scope: the bean is instantiated once per request

- `@juzu.SessionScoped` scope: the bean is instantiated once per session

- `@juzu.FlashScoped` scope: the bean is instantiated once per request but is reused if it was instantiated during an action request in the next render request and only in the first one

6.2.3. Qualifiers

Qualifier are designed to distinguish several instances of a same bean. How does a bean differ from another bean ? it's not really possible to tell, qualifiers simply answer this question, allowing to:

- distinguish beans based upon the qualifier members

- configure the bean instance for a particular usage

The JSR-330 specification provides the `@Named` qualifier whose purpose is to give a name to a bean, for instance

```
@Named("john")
@Inject Person john;

@Named("peter")
@Inject Person peter;
```

6.3. Beans in action

Beans are simply the objects managed by the IOC engine. In a Juzu applications we have several kind of beans:

- Controllers

- Template

- Application beans

- Plugin beans

6.3.1. Template beans

Every template has a corresponding `juzu.template.Template` class at runtime. The template class allows applications to interact with templates, most of the time for rendering purpose:

```
template.render();
```

A template bean is always qualified by the `@Path` qualifier. The path qualifier is simply the value of the path relative to the *templates* package, for instance *index.gtmpl* is a valid qualifier value. The qualifier allows to have several `Template` instances and distinguish them.

Templates have the `@Singleton` scope: a single instance of the template object is created and shared in the IOC container.

6.3.2. Controller beans

Each controller class is turned into a bean, that's how controllers can be injected with other beans. As soon as Juzu finds a class annotated by `@View`, `@Action` or `@Resource`, it is automatically turned into a bean.

Controller have the `Request` scope by default: every time a controller instance is required it will be created for the duration of the request. It is possible to change the scope of a controller by annotating it with another scope annotation managed by Juzu:

```java
@SessionScoped
public class Controller {
    @View
    public void index() { }
}
```

6.3.2.1. Injection a template into a controller

Injecting a template bean into a controller bean is probably the most common Juzu pattern:

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

The template can then be used for rendering purposes:

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

6.3.3. Application beans

Application beans model the custom logic of an application, they are normally injected in controller beans that use them when they process requests. The *binding* plugin allows an application to declare custom beans that can be used in the application.

6.3.3.1. POJO bean binding

Binding a Plain Old Java Object (POJO) is a very simple task to accomplish:

```
@Bindings(@Binding(Mailer.class))
package myapplication;
```

The bean will be entirely managed by the IOC container, the binding plugin will just declare it in the IOC container. The POJO will be created when needed, for instance when it is inserted in a controller.

```
public class MyController {
    @Inject Mailer mailer;
    @Action
    public void sendMail(String recipient, String subject, String message) {
        mail.send(recipient, subject, message);
    }
}
```

6.3.3.2. Abstract bean binding

Binding an abstract class or an interface type is also possible with the `implementation` member of the `@Binding` annotation:

```
@Bindings(@Binding(value=Mailer.class,implementation=MailerImpl.class))
package myapplication;
```

6.3.3.3. Binding with a provider

Sometimes the implementation cannot be created by the IOC container, for instance it may not have a correct constructor, it can only be retrieved using a factory or it should be configured before being used. For such scenarios the implementation can specify a class implementing the `javax.inject.Provider` interface.

```java
public class ConfiguredMailerProvider implements javax.inject.Provider<Mailer>

    private String email
    private String password;

    public ConfiguredMailerProvider() {
        this.email = System.getProperty("mailer.email");
        this.password = System.getProperty("mailer.password");
    }

    public Mailer get() {
        return new MailerImpl(email, password);
    }
}
```

Thanks to the provider, we have a `Mailer` provider that returns a `MailerImpl` configured before usage.

6.3.3.4. Scoped binding

The `@Binding` annotation provides room for declaring a bean scope:

```java
@Bindings(@Binding(value=Mailer.class,scope=Scope.SINGLETON))
```

When the scope is not specified, the scope is determined from the bean or implementation that should be annotated with a scope annotation. When it is specified, it overrides the annotation scope the bean could declare.

6.3.3.5. Qualifying provider

A provider implementation can declare qualifiers on the `get` method they implement in order to set the qualifiers of the returned bean:

```java
public class MailerProvider implements Provider<Mailer> {
    @Named("mailer")
    public Mailer get() {
        return new MailerImpl();
    }
}
```

This is useful for declaring qualifiers on a class that is not annotated by qualifiers, because it is not possible to declare qualifiers in an `@Binding` annotation due to limitations of the Java language.

6.4. Provider factories

Provider factories provides plugability for integrating beans that are not managed by the IOC container. The provider factory is a factory for `javax.inject.Provider` whose purpose is to return a provider for a specific class. Usually provider factories will lookup the service in a registry (like another IOC container) and returns a provider that return them lazily or not.

The provider factory defines the `getProvider` method:

```
/**
 * Returns a provider for a specific type or null if it cannot be produced.
 *
 * @param implementationType the implementation class object
 * @param <T>                the implementation generic type
 * @return a provider for this class or null
 * @throws Exception any exception that would prevent to obtain the provider
 */
<T> Provider<? extends T> getProvider(Class<T> implementationType)
  throws Exception;
```

The factory implementation must provide a public zero argument constructor and it will be instantiated during the application boostrap by Juzu to obtain the provider. The returned providers will then be bound into the IOC container.

The IOC container uses the `java.util.ServiceLoader` discovery mechanism for finding provider factories when injection occurs.

Let's study a simple example with a provider for the current time:

Example 6.2. Time provider factory

```
package my;

public class TimeProviderFactory implements java.inject.ProviderFactory {
  public <T> Provider<? extends T> getProvider(final Class<T> implementationTy
    if (implementationType == java.util.Date.class) {
      return new Provider<T>() {
        public T get() {
          return implementationType.cast(new java.util.Date());
        }
      };
    }
    else {
      return null;
    }
  }
}
```

This provider should be declared in the *META-INF/services/juzu.inject.ProviderFactory* file:

Example 6.3. Time provider configuration

```
my.TimeProvider
```

7

Templating

Templating is the *View* part of a Model View Controlle architecture. We will study in this chapter how the templating system interacts with the Juzu, at compilation time and at runtime, both aspects are very important.

7.1. The templating engines

Juzu can use several templating engines, it provides a native template engine as well as the Mustache templating engine. Those engines are not competing, instead they should be seen as alternatives: the native Groovy engine provides the goodness of the Groovy languages, however sometimes some people prefer logic-less templates and Mustache is a template engine they should use. Let's introduce them briefly.

7.1.1. The native template engine

The native template engine extends the Groovy templating system: it can include snippet of Groovy code or resolve Groovy expressions:

7.1.1.1. Expressions

Expressions are simply Groovy expressions wrapped with the `${...}` syntax:

```
The sky is ${color}
```

7.1.1.2. Scriplets

Groovy code can also literaly be used with the *scriplet* syntax: `<% ... %>`. Within a scriptlet the `out` implicit object can be used for outputting markup:

```
<ul>
<% ["red","green","blue"].each({ color -> out.print("<li>The sky is " + color
</ul>
```

The scriplet syntax `<%= ... %>` can also be used:

```
The sky is <%= color %>
```

7.1.1.3. Controller urls

Controller urls is natively supported by the engine, it allows to create controller URL with a short and compact syntax `@{...}`:

Example 7.1. Controller URL syntax

```
<a href="@{index()}">Home</a>
```

URL expressions can contain parameters and they must be named:

Example 7.2. Controller URL with parameters

```
<a href="@{purchase(product=1)}">Purchase</a>
```

The *purchase* method refers to a controller method, when the application has several controllers, the controller name can be used to prefix the url expression and remove the ambiguity:

Example 7.3. Explicit controller URL

```
<a href="@{Controller.purchase(product=1)}">Purchase</a>
```

Under the hood the controller URL syntax uses the controller compagnion for creating the URL: the `Controller.purchase(product=1)` will uses the controller compagnion `Controller_#purchase(String product)`.

7.1.1.4. Messages

You can resolve a message in resource bundles in a template with the `&{...}` syntax:

Example 7.4. Message syntax

```
<label>&{color}</label>
<input type="text" name="color">
```

When the message is not found, no output will be done. The resource bundle is resolved with the current user locale.

> This feature at the moment only works with portlets that supports resource bundle declaration natively

7.1.1.5. Taglib

The native engine provides taglib support using the `#{tag}...#{/tag}}` or `#{tag/}` syntax:

```
#{title value=Hello/}
```

Available tags are explained in the taglib chapter.

7.1.2. The Mustache template engine

The Mustache template engine uses *logic-less* templates based on Mustache.java the Java port of Mustache. Mustache is very easy to use, you can read the documentation, however we will have a quick overview of how it can be used in Juzu:

7.1.2.1. Variables

Variables uses the {{...}} syntax, they are resolved against template parameters or beans.

```
The sky is {{color}}
```

7.1.2.2. Sections

Mustache sections allows to iterate expressions that are multivalued.

```
todo
```

7.2. Using templates

A template as seen by an application is a bean managed by the IOC container.

7.2.1. Template declaration

Applications use a template by injecting a `juzu.template.Template` object in its controllers qualified by the `juzu.Path` annotation:

Example 7.5. Using a template

```java
public class Controller {

  @Inject
  @Path("index.gtmpl") ❶
  Template index;

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

❶   Declares the template path

❷   Renders the template and send the markup to the response

The `juzu.Path` annotation is a qualifier annotation managed by the IOC container. It is very similar to the `@javax.inject.Named` qualifier, but it has a special meaning for Juzu for declaring the template.

The `render` method of a template returns a `juzu.Response.Content` response which can also be returned by the controller method. This is equivalent to the previous example.

Example 7.6. Returning the generated juzu.Response.Content

```java
@View
public Response.Content index() {
  return index.render();
}
```

7.2.2. Type safe parameters

Template type safe parameters brings more type safety in your applications. Templates can declare parameters and they are made available on a subclass of the `juzu.template.Template` class.

Parameters are declared using the taglib support of the native template engine

Example 7.7. Native template parameter declaration

```
#{param name=color/}
The sky is ${color}.
```

or the pragma support of the Mustache engine

Example 7.8. Mustache template parameter declaration

```
{{%param color}}
The sky is {{color}}.
```

When the template is declared in a controller, a subclass of `juzu.template.Template` can be used:

```
package weather;

public class Controller {

  @Inject
  @Path("sky.gtmpl")
  weather.templates.sky sky; ❶

  @View
  public void index() {
    sky.with().color("blue").render(); ❷
  }
}
```

❶   The `weather.templates.sky` typed template class

❷   Use the `sky` template `color` parameter

The `weather.templates.sky` class does not exist in the original source but it is available when the application is compiled because it will be generated by Juzu compiler integration. The `sky` templates provides a *fluent* syntax to bind parameters: `sky.with().color("blue").render()`.

7.2.3. Expression resolution

When we studied the templating engine syntax but we did not mentioned exactly how expression are resolved.

7.2.3.1. Single name expressions

Both templating system provides a syntax for resolving single name expressions:

- `${...}` for Groovy

- `{{...}}` for Mustache

Resolution is performed against template parameters or bean named with the `javax.inject.Named` qualifier.

Example 7.9. Named bean

```
@javax.inject.Named("color")
public class Color {
  public String toString() {
    return "red";
  }
}
```

Example 7.10. Template parameters

```
index.with().set("color", "red").render(); ❶
index.with().color("red").render(); ❷
```

❶   Detyped version

❷   Type safe version

7.2.3.2. Compound expressions

Compound expressions are resolved the same way for the first name and the expression resolve will attempt to navigate the rest of the expressions from this object:

- `${weather.color}` for Groovy

- `{{#weather}}{{color}}{{/weather}}` for Mustache

Example 7.11. Named bean

```java
@javax.inject.Named("weather")
public class Weather {

  private String color;

  public Weather(String color) {
    this.color = color;
  }

  public Weather() {
    this.color = "red";
  }

  public String getColor() {
    return color;
  }
}
```

Example 7.12. Template parameters

```
index.with().set("weather", new Weather("blue")).render(); ❶
index.with().color(new Weather("blue")).render(); ❷
```

❶   Detyped version

❷   Type safe version

}}}

8

Templating SPI

This chapter dives into the template life cycle from the compilation time to the run time. We will describe the template Service Provider Interface (SPI), the SPI is designed to make Juzu templating extensible and integrating template engines in Juzu. This chapter is optional is you are only writing ab application with Juzu, however it is a must read if you want to know more Juzu internals or if you want to understand how to integrate a template engine in Juzu.

When a Juzu application is compiled, the Juzu annotation processor detects the `@Path` annotations and triggers the compilation of the related templates. The template compilation can be split in two parts:

- Generating the template companion class that inherits the `juzu.template.Template` class. This part is generic and works with any templating system, it is entirely managed by Juzu.

- Processing the template file, this task is delegated to the `TemplateProvider` interface and is extensible. The provider allows to have several templating system in Juzu and decouples the template compilation process from the details of the templating engine.

8.1. Compiling a Groovy template

Let's study an example with the Groovy template at compilation time.

Figure 8.1. Compiling a Groovy template
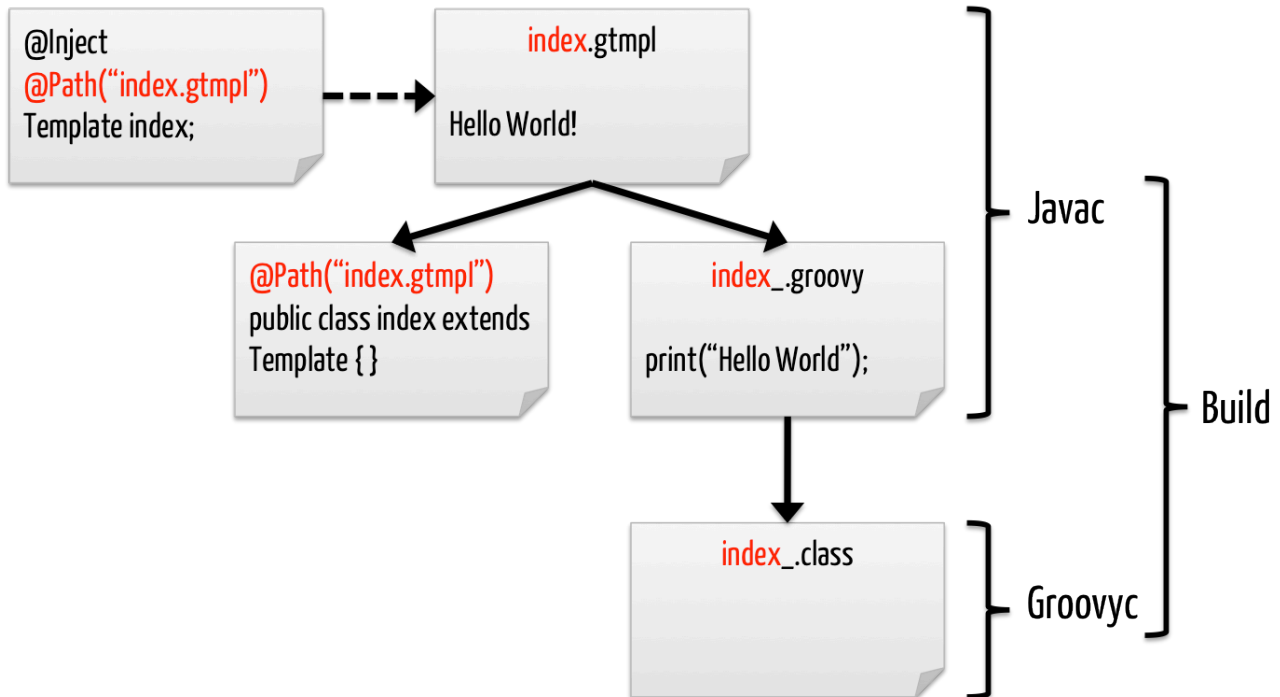
When the Java compiler is invoked, the following steps are executed

1. The Java compiler triggers the Juzu annotation processor when it finds the `@Path` annotation

2. Juzu resolves the relative path to the `templates` package of the application

    1. When the template cannot be resolved a compilation error is triggered

    2. Otherwise the template is loaded

3. The template provider is looked up according to the file name extension, it generates the *index_.groovy* source file

4. Juzu creates the `index` class that extends the `juzu.template.Template` class annotated by the `@Path("index.gtmpl")` annotation

After that the only remaining part is to compile the *index_.groovy* source to a class. It can be achieved either at build time using the *groovyc* compiler or at load time when the `index` template is loaded using a `GroovyClassLoader`. The former approach makes the build a bit more complex (but not much as Groovy compilation is fairly well supported in build systems or IDEs) as it requires to run a Groovy compilation but it will perform additional validation of the template as well as reduce the load time of the template. The later approach will detect any compilation error (such as Groovy syntax error) at runtime and the *index_.groovy* compilation will take a few milliseconds.

This flexibility allows to use the lazy approach during development and when the application is released then the Groovy compiler can be used to compile the *index_.groovy*.

8.2. Type safe URL resolution

Groovy templates provides the @{...} syntax for generating URL from the application controllers. This section gives an overview of the underlying resolution mechanism.

Figure 8.2. Template URL resolution during compilation

- Parse: the template is parsed into its model representation

- Resolve: the `index` link is resolved againt the controller meta model

- Validate: the `index` link is validated

- Emit: the corresponding *index_.groovy* file is emitted and save on the class output

- Compile: the Groovy source is compiled into a class by the *groovyc* compiler (this part is done after *javac*)

8.3. Template Service Provider Interface

Juzu provides a Service Provider Interface (SPI) for integrating thirdparty template engine. Actually all template system are integrated with the SPI. We will study briefly the integration points so you can integrate a template engine of your choice in Juzu.

8.3.1. Template providers

The `juzu.impl.template.spi.TemplateProvider` is the main entry point when a templating system is integrated. The provider is triggered during the compilation phase by the APT system built into the Java compiler.

```java
/**
 * A provider for templating system.
 *
 * @author <a href="mailto:julien.viet@exoplatform.com">Julien Viet</a>
 * @param <M> the template model
 */
public abstract class TemplateProvider<M extends Serializable> {
  ...
}
```

The provider must declare the template model `<M>` generic type, this type must be a serializable as Juzu will sometimes write template models on the disk during the compilation. This usually happens only in Eclipse due its incremental compiler architecture. The type specified by the provider is privately managed (i.e it is opaque for Juzu) and it symbolizes an internal representation of the parsed source (usually an Abstract Syntax Tree), it will be used in various methods of the provider.

Let's have a review of the methods of this class to have a better understanding.

```java
public abstract String getSourceExtension();
```

The `getSourceExtension()` method is used to determine what file extension the provider can compile. The implementation should return a constant value, for instance the Groovy provide simply returns the `gtmpl` value.

```java
public abstract M parse(
    ParseContext context,
    CharSequence source) throws TemplateException;

public abstract void process(
    ProcessContext context,
    Template<M> template) throws TemplateException;

public abstract void emit(
    EmitContext context,
    Template<M> template) throws TemplateException, IOException;
```

The `parse`, `process` and `emit` methods care about transforming the template source to its final representation: the compiled template.

- The `parse` method is invoked with the content of the template and returns a template model. The representation returned by the parse method is a parsed representation of the template source. If a parsing error occurs the method can throw a `TemplateException`.

- The `process` method is invoked after the template is parsed with the necessary context for performing further processing of the template, for instance the Groovy templating engine performs the resolution of type safe URLs or type safe parameters declaration at this moment. During the process:

  - The provider can resolve other templates using the `ProcessContext`, if the template

- to resolve is not yet loaded it will trigger the `parse`/`process`/`emit` lifecycle, it if was already processed the template is simply returned

    - The implementation can resolve controller methods and translate them into method invocation, this is used for checking type safe URL and translating them into controller companion invocation

    - The `juzu.impl.template.spi.Template` argument represents the template, it has several fields such as the template model or the template path

    - The implementation can declare type safe parameters using the `Template#addParameter(String)` method. The declared parameters will be generated on the `juzu.template.Template` subclass

- The `emit` method is invoked when the template processing is over. The `EmitContext` interface can be used to create resources during this round.

```
public abstract Class<? extends TemplateStub> getTemplateStubType();
```

Finally the `getTemplateStubType()` returns the type of a java class that will be used for creating a template stub. For each template, a stub is created, the stub is responsible for loading the template at runtime, i.e the original template or the compiled template that may have been generated during compilation during the `emit` callback.

8.3.2. Template stub

Template stubs are java classes created by Juzu for managing a template at runtime on behalf of the provider. Each provider provides its own stub implementation as a `juzu.impl.template.spi.TemplateStub` subclass.

A stub must provide a public constructor accepting a `java.lang.String` argument: the template id. The template id if the class name of the generated template. In addition, a stub must implement two abstract methods:

```
protected abstract void doInit(ClassLoader loader);

protected abstract void doRender(TemplateRenderContext renderContext)
    throws TemplateExecutionException, IOException;
```

The `doInit` method loads the template using the provided `ClassLoader`, it will be call only once before the template is rendered. Usually it uses the template id provided during the construction of the template to locate the template on the disk, in its original form or in its compiled form.

The `doRender` method renders the template using the provided `TemplateRenderContext`. The render context provides the necessary hooks such as:

- Producing markup

- Setting the title

- Obtaining the locale

- Accessing parameters or application beans for resolving expressions

8.4. Template at work

After having described the various pieces of the templating SPI, let's look at how the template generated stubs are used by Juzu templating system at runtime.

When the controller declares the *index.gtmpl* template the compiler produces three artifacts

- the `index` class template inherits `juzu.template.Template`: it is the only class visible from the controller and the whole application

- the *index_.groovy* Groovy template is the effective template code: it produces the markup, resolve expressions, etc...

Figure 8.3. index groovy at work
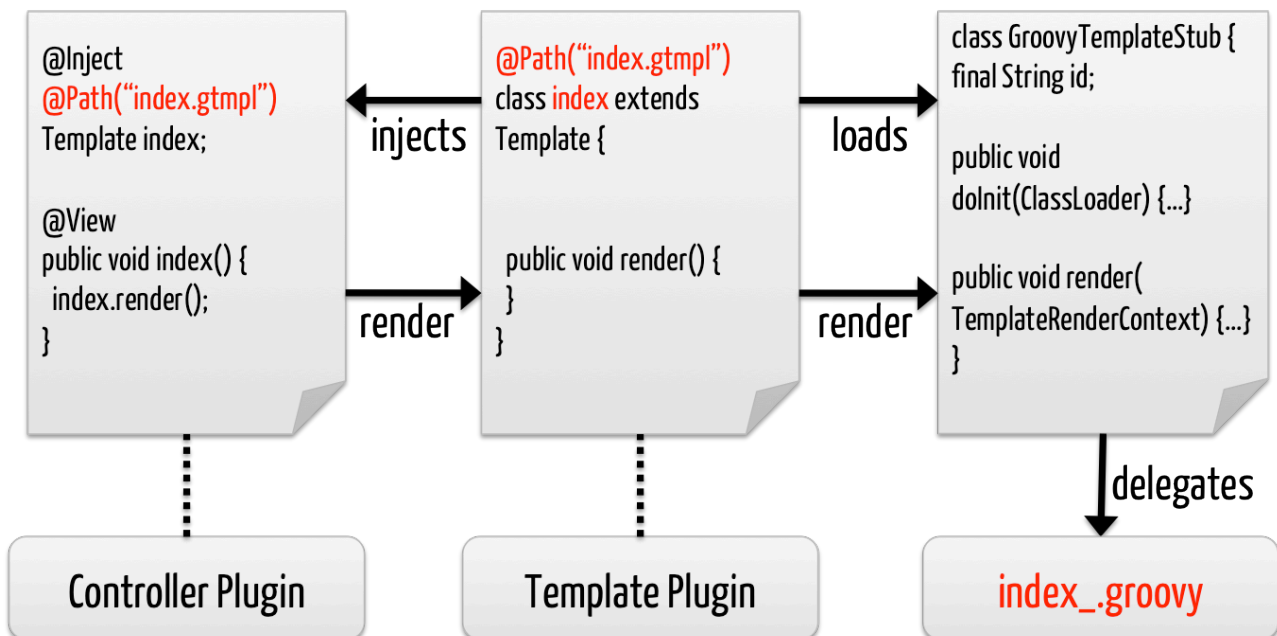
When a controller is instantiated, the `index` template instance is injected into the controller, the `@Path` annotation plays an essential role because it's a qualifier and that qualifier is used to distinguish the correct subclass to inject in the controller.

When the template is created, the corresponding template stub is instantiated. When the template needs to be rendered, the `doInit(ClassLoader)` method of the stub is invoked. At this moment the Groovy `index_` class is loaded from the class loader, when the class is not found, the `index_.groovy` is loaded and it is compiled on the fly.

8.5. Qualified template class

Controller can be injected with the `juzu.template.Template` class, however they can also be injected with the template subclass that was genereted by Juzu: instead of using the qualified template injection, the controller declares the template *index* subclass. This approach cab be used when type safe parameters are used as only the `index` type declares the fluent API.

For instance if the *index.gtmpl* declares the *color* parameter the `index` class will look like:

```
@Path("index.gtmpl")
public class index extends Template {

  ...

  public index with() {
    return new index.Builder();
  }

  public class Builder extends Template.Builder {

    public Builder color(String color) {
      // Generated code
    }
  }
}
```

The controller can then use the fluent API:

```
public class Controller {

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

  @View
  public void index() {
    index.with().color("red").render();
  }
}
```

9

Taglib

A tag library is an essential component of a templating system, allowing to enrich a templating with encapsulated programmable logic.

> Juzu does not yet allow application to define their own tags, it will be added a a new feature in a future version.

9.1. Taglib syntax

Like most taglib syntaxes, Juzu provides two syntaxes for invoking a tag:

Example 9.1. Start and end tag syntax

```
#{foo}bar#{/foo}
```

The start/end syntax opens the tag with `#{foo}` and ends it with `#{/foo}`.

A tag can also be empty:

Example 9.2. Empty tag syntax

```
#{foo/}
```

A tag can also be invoked empty with the `#{foo/}` syntax.

9.2. Include tag

The *include* tag simply includes a template inside the current template. The inclusion is dynamic and not static, meaning that the content of the included template is not *inserted* in the calling template, instead when inclusion is performed the control is passed to the included template.

Example 9.3. The include tag

```
#{include path=dispatched.gtmpl/}
```

The *path* attribute determines the template to include, the path value is relative to the templates package.

9.3. Decorate / Insert tag

The *decorate* tag allows the content of the decorating template to wrap the content of the template invoking the tag. The *insert* tag should be used in the decorating template to specify the place where to insert the markup produced by the template to decorate.

Example 9.4. The wrapped template

```
#{decorate path=box.gtmpl/}
```

Example 9.5. The decoraring template

```
<div style="border: 1px solid black">
#{insert/}
</div>
```

9.4. Title tag

The *title* tag specifies a title to insert in the `juzu.Response.Content` object the template will produce.

Example 9.6. Setting the title

```
#{title value=Home/}
```

9.5. Param tag

The *param* tag enhances the type safety of templates, allowing to declare parameters for executing a template. When such a parameter is declared, the generated template class companion will have a fluent parameter for setting the value of the parameter:

Example 9.7. Declaring a template parameter

```
#{param name=color/}
```

Example 9.8. Using the template parameter

```
@Inject my.templates.index index;

@View
public void index() {
  index.with().color("red").render();
}
```

10

Assets

Web assets are resources used over the web such as stylesheet and script files.

10.1. Asset serving

Assets are declared by the application with the following attributes:

- an *id* to reference it

- a *location* defining where to find the asset

- a *source* for locating the asset

- a list of *dependencies* referencing other assets specifying the assets that needs to be served before the declaration

When an application is deployed, assets are registered with the *asset manager*. The asset manager has several responsibilities:

- manage asset dependencies: the order in which assets are literaly declared when they are served. For instance the *jquery-ui* asset depends on the *jquery* asset because the jquery script must be loaded before the *jquery-ui* script.

- resolve asset references: each asset reference must be resolved and produce a final web url that will produce the resource when it is resolved by the web browsers

During a request, asset identifiers are added to the response. At the end of the request, the runtime uses the asset manager to translate the response assets into a list of uri to add to the page.

Figure 10.1. Using assets in a request
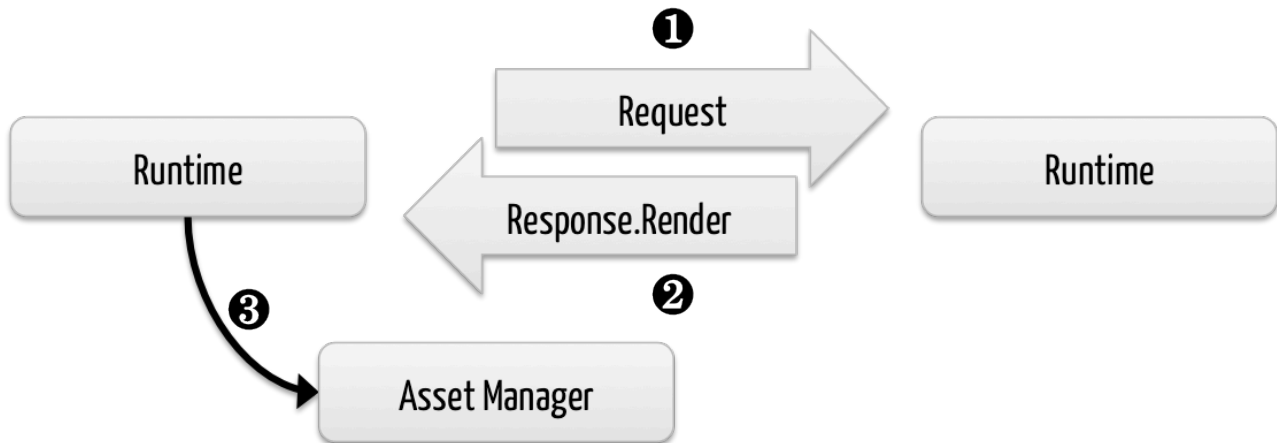
An asset reference is a link to an asset value that is configured externally, thus an asset of any kind will always resolve to a location and an uri. Let's examine the different possible asset location:

- `AssetLocation.URL`: the value is opaque to Juzu, for instance the a CDN hosted script such as *https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js*.

- `AssetLocation.SERVER`: the asset is served by the same web server in which Juzu is deployed. If the asset value is relative, the final uri will resolve relatively to the web archive context address.

- `AssetLocation.APPLICATION`: the asset is served by Juzu *asset server* (a servlet configured in the web application) and the resource is located on the classpath.

10.2. Asset server

For serving classpath assets, Juzu requires the configuration of the asset server as a servlet declaration:

```
<servlet>
  <servlet-name>AssetServlet</servlet-name>
  <servlet-class>juzu.impl.asset.AssetServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>AssetServlet</servlet-name>
  <url-pattern>/assets/*</url-pattern>
</servlet-mapping>
```

This declaration should be in the *web.xml* of the application whether it is a servlet or a portlet application.

> If you are using Servlet 3.0, this declaration is not necessary

10.3. Asset plugin

The asset plugin provides declarative asset configuration, it will take care of registering assets on the asset manager. The `@Assets` annotation declares a list of assets used by the an application.

Example 10.1. JQuery UI declarative asset configuration

```
@Assets(
  scripts = {
    @Script(id = "jquery",
            src = "javascripts/jquery-1.7.1.min.js"), ❶
    @Script(src = "javascripts/jquery-ui-1.7.2.custom.min.js", ❷
            depends = "jquery") ❸
  },
  stylesheets = {
    @Stylesheet(src = "ui-lightness/jquery-ui-1.7.2.custom.css")
  }
)
package my.application;
```

❶   Declares the jquery asset

❷   Declares the jquery-ui asset

❸   Make jquery-ui depend on jquery

The assets will be served by the asset servlet because the application location is the default when not specified. The first `@Script` annotation declares the JQuery asset reference identified by the `id` member as being *jquery*. The second `@Script` annotation declares the JQuery-UI plugin, it does not need an *id* member because nothing refers to it, however it declares a dependency with the *depends* member that declares it depending on the *jquery* asset. JQuery-UI requires also a stylesheet to be served along with the script, it is achieved thanks to the `@Stylesheet` annotation.

10.3.1. Application assets

Applications assets can be located anywhere on the application classpath, they can be either absolute or relatives. Relative assets declared by the asset plugin must be located in the `assets` package of the application, for instance an application packaged under `my.application` will have its relative assets located under `my.application.assets`.

Example 10.2. Declarative relative application asset configuration

```
@Assets(scripts = @Script(
  src = "myscript.js"))
package my.application;
```

The location `AssetLocation.APPLICATION` is not declared because it is the default one.

10.3.2. Server assets

Server assets are served by the webserver in which the application is deployed. Relative server assets are served from the war file containing the application.

Example 10.3. Declarative relative server asset configuration

```
@Assets(scripts = @Script(src = "myscript.js", location = AssetLocation.SERVER
package my.application;
```

10.3.3. External assets

External assets declares an opaque URL for Juzu.

Example 10.4. External classpath asset configuration

```
@Assets(scripts = @Script(
  src = "https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js",
  location = AssetLocation.CLASSPATH))
package my.application;
```

10.4. Managing assets programmatically

The `@Assets` annotations will serve all the declared assets in `scripts` and `stylesheets`. Sometimes it is convenient to declared assets but serve them upon application request. This can be achieved with the `declaredScripts` and `declaredStylesheets` declarations:

```
@Assets(declaredScripts = @Script(
  id = "myscript",
  src = "myscript.js"))
package my.application;
```

When an application requires an asset, it adds the asset to the `Response.Content` object:

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

@View
public Response.Content index() {
  Response.Content render = index.render();
  render.withScript("myscript");
  return render;
}
```

The same with a fluent syntax:

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

@View
public Response.Content index() {
  return index.render().withScript(myscript""));
}
```

11

Javascript modularity

The AMD plugin provides declarative support for JavaScript modules using annotations. It relies on the Asynchronous Module Definition specification implemented by the RequireJS project.

11.1. Introduction to modules

JavaScript does not provide a natural way for namespacing, the notion of module was designed to solve this problem. This natural lack of namespacing can be perceived as a lack, instead it should be seen as an advantage as modules provide namespacing and more: indeed the module pattern allows to create dependencies between modules and resolve them at runtime enabling on demand and parallel loading of JavaScript resources.

This guide will not explain modules because we haven't designed a module system for Juzu. Instead Juzu uses the RequireJS library and integrates it. Therefore the best documentation you can read about modules is the RequireJS documentation you can also read the excellent article about modules in depth.

In the essence the notion of module can be viewed as:

- An identifier

- A list of dependencies on the modules required by the module to work properly

- The code packaged usually expressed as a self-executing function

- The product which is an object produced by the module that is usually consumed by other modules

At runtime the dependency system defines a graph of function to execute, the product of each module being injected in the other modules. It can be seen as a simple dependency injection system able to load modules in an asynchronous and parallel fashion providing parallel loading, namespacing and dependency management.

11.2. Defining a module

The `@Defines` and `@Define` are used to declare the JavaScript which is either a canonical module as defined by the AMD specification or a simple self-executing function.

When the javascript is self-executing function, this script is wrapped to provide the expected RequireJS format, let's study a quick example with a module:

Example 11.1. Defining a simple module

```
@Defines(value={@Define(name="Foo", path="foo.js")})
package my.application
```

Example 11.2. The ##foo.js## self-executing function

```
(function () {
  //Do something
})();
```

Equivalent to this declaration in the RequireJS format:

```
define("Foo", function() {
  return //Something
});
```

Modularity allows to define cross dependencies between modules, the `@Define` annotation allows that:

Example 11.3. Defining a module with dependencies

```
@Defines(
  value = {
    @Define(name="Foo", path="foo.js"),
    @Define(name="Bar", path="bar.js", dependencies={@Dependency(name="Foo", a
)
```

Example 11.4. The ##bar.js## self-executing function

```
(function(foo) {
  //Do something
})(foo);
```

Equivalent to this declaration in the RequireJS format:

```
define("Bar", ["Foo"], function(foo) {
  return //Something
});
```

The Juzu AMD plugin allows to provide custom adapter for adapting the script to the expected format. With this flexibility it is possible to declare an adapter wrapping the adapted script. The jQuery library is a good example of how a custom adapter can be useful. Thanks to the adapter feature we can reuse the jQuery without any change:

Example 11.5. The jQuery uses the following construct for defining itself

```
(function(window, undefined) {
})(window);
```

The main issue with this construct is that it will bind jQuery to the window but most importantly it will not return any value as expected by the dependency system. Thanks to the custom adapter we can integrate it easily:

```
@Define(
  id = "jquery",
  path="jquery-1.7.1.js",
  adapter="(function() { @{include} return jQuery.noConflict(true);})();")
```

The adapter attribute can contains mixed content and the `@{include}` will perform a mere inclusion (as a C language includes) of the original jQuery script in the resulting module:

```
define("jquery", [], function() {
  return (function() {
    (function(window, undefined) {
    })(window);
    return jQuery.noConflict(true);
  })();
});
```

The Juzu AMD definition adapts only the javascript located at `AssetLocation.APPLICATION`. The `@Require` annotation must be used for other kind of locations.

11.3. Requiring a module

The Juzu AMD `@Require` annotation only register a JavaScript module for RequireJS Loader. It does not wrap or adapt content and does not manage module dependencies.

Here is an example of using `@Requires` and `@Require` to register the `foo.js` script and the `bar.js` scripts that depends on `foo.js`.

```
@Requires (
  value = {
    @Require(id = "Foo", path = "foo.js"),
    @Require(id = "Bar", path = "bar.js")}
)
package my.application;
```

Example 11.6. The ##foo.js## module declaration

```
define("Foo", function() {
  return {
    text: "Hello"
  };
});
```

Example 11.7. The ##bar.js## module declaration

```
define("Bar", ["Foo"], function(foo) {
  return {
    text : foo.text + " World"
  };
});
```

12

Juzu Servlet Plugin

The servlet plugin enhance Juzu servlet applications.

12.1. Servlet class generation

A Juzu servlet application is managed by a `JuzuServlet` configured with the application name. Since Servlet 3.0, configuration can be easier thanks to servlet annotations. Juzu leverages this capability and is able to generate a servlet for an application with the `juzu.plugin.servlet.Servlet` annotation:

Example 12.1. Juzu servlet generation

```
@Application
@Servlet("/") ❶
package my.application;
```

❶   The application *url-pattern*

12.2. Asset server automatic registration

The jar of the servlet plugin contains a *web-fragment.xml* that automatically declares the asset servlet simplifying further more the configuration of the application.

13

Juzu Portlet Plugin

The portlet plugin enhance Juzu portlet applications.

13.1. Portlet class generation

A Juzu portlet application is managed by a `JuzuPortlet` configured with the application name. The `@juzu.plugin.portlet.Portlet` annotation can be used to generate a subclass of the `JuzuPortlet` that configures the application name for you, easing the configuration of the *portlet.xml* corresponding section.

```
@Portlet
package my;
```

```
<portlet>
  <portlet-name>MyApplication</portlet-name>
  <display-name xml:lang="EN">My Application</display-name>
  <portlet-class>myapp.MyPortlet</portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <portlet-info>
    <title>My Application</title>
  </portlet-info>
</portlet>
```

The plugin will generate the portlet using the application name with the first letter capitalized and the *Portlet* suffix. In our example the *my* application generates the `MyPortlet` class. If you don't like it you can change the name of the generated class in the application:

```
@Portlet(name "MyGreatPortlet")
package my;
```

```
<portlet>
  <portlet-name>MyApplication</portlet-name>
  <display-name xml:lang="EN">My Application</display-name>
  <portlet-class>myapp.MyGreatPortlet</portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <portlet-info>
    <title>My Application</title>
  </portlet-info>
</portlet>
```

13.2. Portlet preferences injection

During the various phase of an application, the current portlet preferences can be injected::

Example 13.1. Injecting portlet preferences

```
@Inject javax.portlet.PortletPreferences preferences;
```

The same restriction defined in the portlet specification applies to the provided preferences object: i.e saving preferences can only be performed during an action phase.

13.3. Resource bundle injection

During the various phase of an application, the portlet resource bundle for the current locale can be injected:

Example 13.2. Injecting the portlet resource bundle

```
@Inject java.util.ResourceBundle bundle;
```

This is equivalent of doing:

```
Locale locale = request.getLocale();
ResourceBundle bundle = portlet.getConfig().getResourceBundle(locale);
```

This resource bundle can be configured in the *portlet.xml* deployment descriptor.

13.4. Building

Add the Portlet plugin jar to your compilation classpath.

In Maven it can achieved by adding the Less plugin dependency to your POM:

```
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-plugins-portlet</artifactId>
  <version>0.7.0-beta15</version>
</dependency>
```

14

Juzu File Upload Plugin

The file upload plugin integrates Apache Commons FileUpload in Juzu. The plugin decodes multipart requests as file objects and can inject them as controller method parameters. This plugin works with the servlet bridge and the portlet bridge.

14.1. File upload in an action phase

File upload can be handled during an action of a portlet or a servlet:

```
@Action
@Route("/upload")
public void upload(org.apache.commons.fileupload.FileItem file) {
    if (file != null) {
        // Handle the file upload
    }
}
```

> The `@Route` annotation is only meaningfull for the servlet bridge. In case of a portlet, the URL is managed by the portal.

14.2. File upload in a resource phase

File upload can also be handled in a resource phase.

```
@Resource
@Route("/upload")
public Response.Content upload(org.apache.commons.fileupload.FileItem file) {
    if (file != null) {
        // Handle the file upload
    }
    return Response.ok("Upload is done");
}
```

Handling upload in a resource phase can be used when the file is uploaded via Ajax: the application does not want a view phase to be triggered after the upload.

15

Juzu Less Plugin

LESS is a dynamic stylesheet language which extends CSS with dynamic behavior such as variables, mixins, operations and functions. LESS is easy to learn thanks to the online documentation.

Juzu provides a LESS plugin that takes care of compiling a LESS stylesheet into a CSS stylesheet which are then served by the Asset plugin. This chapter explains how to use LESS and combine it with the Asset plugin.

15.1. Usage

The LESS plugin operates at compilation time only because the only task he has to do is to transform a LESS source code into a CSS stylesheet. The runtime part is usually done by the Asset plugin.

The `@Less` annotation annotates a package containing an `assets` package. This `assets` package should contain the LESS files to be compiled.

Example 15.1. Annotating an application package for processing LESS files

```
@Less("stylesheet.less")
@Application
package myapp;

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

The *stylesheet.less* file will be located in the `myapp.assets` package. The `assets` child package of the annotated package should contain the stylesheet, this is done on purpose to coincide exactly with the `assets` package used by the Asset plugin. During the compilation phase the *stylesheet.less* will be compiled to the *stylesheet.css*. If we want this file to be served with the application we simply add the corresponding `@Assets` annotation:

Example 15.2. LESS and Asset plugins in action

```
@Less("stylesheet.less")
@Assets(stylesheets = @Stylesheet(
  src = "stylesheet.css",
  location = AssetLocation.CLASSPATH)
)
@Application
package myapp;

import juzu.Application;
import juzu.asset.AssetLocation;
import juzu.plugin.less.Less;
import juzu.plugin.asset.Assets;
import juzu.plugin.asset.Stylesheet;
```

By default LESS will use a default formatting for the generated CSS. To achieve smaller CSS size, a *minify* option can be used, this option will trim the whitespace when processing the file : `@Less(value = "stylesheet.less", minify = true)`.

15.2. Building

Add the Less plugin jar to your compilation classpath.

In Maven it can achieved by adding the Less plugin dependency to your POM:

```
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-plugins-less</artifactId>
  <version>0.7.0-beta15</version>
</dependency>
```

16

Juzu WebJars Plugin

WebJars are client-side web libraries (e.g. jQuery & Bootstrap) packaged into jar files. WebJars allow to declaratively set the version, use a consistent version across an application, and easily deal with transitive dependencies.

Juzu provides a WebJars plugin that copies resources in jar libraries to application assets and then served by the *Asset* plugin or the *AMD* plugin.

16.1. Usage

Example 16.1. Annotating an application package for declaring WebJars

```
@Application
@WebJars("foo.js")
package myapp;

import juzu.Application;
import juzu.plugin.webjars.WebJars;
```

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>foo</artifactId>
  <version>1.0</version>
</dependency>
```

The *foo.js* file will be located in the `myapp.assets` package.

If we want this file to be served with the application we simply add the corresponding `@Assets` annotation:

Example 16.2. WebJars and Asset plugin in action

```
@Application
@WebJars("foo.js")
@Assets(scripts = {
  @Script(
    id = "foo",
    src="foo.js"
  )
})
package myapp;

import juzu.Application;
import juzu.plugin.asset.Assets;
import juzu.plugin.asset.Script;
import juzu.plugin.webjars.WebJars;
```

It can also be used as a JavaScript module with the *AMD* plugin:

Example 16.3. WebJars and AMD plugin in action

```
@Application
@WebJars("foo.js")
@Defines({
  @Define(
    name = "Foo",
    path="foo.js"
  ),
  @Define(
    name = "Bar",
    path="bar.js",
    dependencies = {@Dependency(name = "Foo")}
  )
})
package myapp;

import juzu.Application;
import juzu.plugin.amd.Define;
import juzu.plugin.amd.Defines;
import juzu.plugin.amd.Dependency;
import juzu.plugin.webjars.WebJars;
```

16.2. Building

Add the WebJars plugin jar to your compilation classpath.

In Maven it can achieved by adding the WebJars plugin dependency to your POM:

```
<dependency>
  <groupId>org.juzu</groupId>
  <artifactId>juzu-plugins-webjars</artifactId>
  <version>0.7.0-beta15</version>
</dependency>
```