Seam Faces Module

Reference Guide

Brian Leathem

Lincoln Baxter III

Dan Allen

Nicklas Karlsson

Introduction

The goal of Seam Faces is to provide a fully integrated CDI programming model to the JavaServer Faces (JSF) 2.0 web-framework. With features such as observing Events, providing injection support for life-cycle artifacts (FacesContext, NavigationHandler,) and more.

Installation

To use the Seam Faces module, you need to put the API and implementation JARs on the classpath of your web application. Most of the features of Seam Faces are enabled automatically when it's added to the classpath. Some extra configuration, covered below, is required if you are not using a Servlet 3-compliant container.

1.1. Maven dependency configuration

If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following single dependency to your pom.xml file to include Seam Faces:

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

Tip

Substitute the expression ${seam.faces.version} with the most recent or appropriate version of Seam Faces. Alternatively, you can create a *Maven user-defined property* to satisfy this substitution so you can centrally manage the version.

Alternatively, you can use the API at compile time and only include the implementation at runtime. This protects you from inadvertantly depending on an implementation class.

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

<dependency>
  <groupId>org.jboss.seam.faces</groupId>
  <artifactId>seam-faces-impl</artifactId>
  <version>${seam.faces.version}</version>
  <scope>runtime</scope>
```

```
</dependency>
```

In a Servlet 3.0 or Java EE 6 environment, *your configuration is now complete!*

1.2. Pre-Servlet 3.0 configuration

If you are using Java EE 5 or some other Servlet 2.5 container, then you need to manually register several Servlet components in your application's web.xml to activate the features provided by this module:

```
<listener>
    <listener-class>org.jboss.seam.faces.beanManager.BeanManagerServletContextListener</
listener-class>
</listener>
```

You're now ready to dive into the JSF enhancements provided for you by the Seam Faces module!

1.3. How to setup JSF in a Java EE 6 webapp

Seam Faces requires a working JSF 2.0 configuration. To get a working JSF 2.0 environment in a Java EE 6 environment, you need one of the following:

1. Bundle the seam-faces jar in your web-app (this sets up jsf for you)
2. if not #1, you need empty faces-config.xml, where the root element must be present.
3. if not #1 or #2, you need a web.xml with the Faces Servlet defined.

The *JBoss JSF documentation* [http://docs.jboss.org/jbossas/6/JSF_Guide/en-US/html/jsf.deployer.config.html] provides further details on #2 and #3 above, but these steps are unnecessary when you use Seam Faces (#1). this is beacuse Seam Faces scans for the presence of the Seam Servlet, and programatically registers it for you if it's not present.

Faces Scoping Support

JSF 2.0 introduced the concept of the Flash object and the @ViewScope; however, JSF 2.0 did not provide annotations accessing the Flash, and CDI does not support the non-standard ViewScope by default. The Seam Faces module does both, in addition to adding a new @RenderScoped context. Beans stored in the Render Scope will survive until the next page is rendered. For the most part, beans stored in the ViewScope will survive as long as a user remains on the same page, and data in the JSF 2 Flash will survive as long as the flash survives).

2.1. @RenderScoped

Beans placed in the @RenderScoped context are effectively scoped to, and live through but not after, "the next Render Response phase".

You should think about using the Render scope if you want to store information that will be relevant to the user even after an action sends them to another view. For instance, when a user submits a form, you may want to invoke JSF navigation and redirect the user to another page in the site; if you needed to store a message to be displayed when the next page is rendered -but no longer- you would store that message in the RenderContext. Fortunately, Seam provides RenderScoped messages by default, via the *Seam Messages API*.

To place a bean in the Render scope, use the `@org.jboss.seam.faces.context.RenderScoped` annotation. This means that your bean will be stored in the `org.jboss.seam.context.RenderContext` object until the next page is rendered, at which point the RenderScope will be cleared.

```
@RenderScoped
public class Bean {
   // ...
}
```

@RenderScoped beans are destroyed when the next JSF RENDER_RESPONSE phase ends, however, if a user has multiple browser windows open for the same user-session, multiple RenderContexts will be created, one for each incoming request. Seam Faces keeps track of which RenderContext belongs to each request, and will restore/destroy them appropriately. If there is more than one active RenderContext at the time when you issue a redirect, you will see a URL parameter "?fid=..." appended to the end of the outbound URL, this is to ensure the correct context is restored when the request is received by the server, and will not be present if only one context is active.

Caution

If you want to use the Render Scope with custom navigation in your application, be sure to call `ExternalContext.encodeRedirectURL(String url, Map<String, List<String>> queryParams)` on any URL before using it to issue a redirect. This will ensure that the RenderContext ID is properly appended to the URL, enabling the RenderContext to be restored on the subsequent request. This is only necessary if issuing a Servlet Redirect; for the cases where Faces non-redirecting navigation is used, no URL parameter is necessary, and the context will be destroyed at the end of the current request.

2.2. @Inject javax.faces.context.Flash flash

JSF 2 does not provide proper system events to create a functional `@FlashScoped` context annotation integrated with CDI, so until a workaround can be found, or JSF 2 is enhanced, you can access the Flash via the @Inject annotation. For more information on the *JSF Flash* [https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/context/Flash.html], read *this API doc* [https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/context/Flash.html].

```
public class Bean {
  @Inject private Flash flash;
  // ...
}
```

2.3. @ViewScoped

To scope a bean to the View, use the `@javax.faces.bean.ViewScoped` annotation. This means that your bean will be stored in the `javax.faces.component.UIViewRoot` object associated with the view in which it was accessed. Each JSF view (faces-page) will store its own instance of the bean, just like each HttpServletRequest has its own instance of a @RequestScoped bean.

```
@ViewScoped
public class Bean {
   // ...
}
```

Caution

@ViewScoped beans are destroyed when the JSF UIViewRoot object is destroyed. This means that the life-span of @ViewScoped beans is dependent on the `javax.faces.STATE_SAVING_METHOD` employed by the application itself, but in general one can assume that the bean will live as long as the user remains on the same page.

Messages API

While JSF already has the concept of adding `FacesMessage` objects to the FacesContext in order for those messages to be displayed to the user when the view is rendered, Seam Faces takes this concept one step farther with the Messages API provided by the Seam International module. Messages are template-based, and can be added directly via the code, or templates can be loaded from resource bundles using a `BundleKey`.

3.1. Adding Messages

Consistent with the CDI programming model, the Messages API is provided via bean injection. To add a new message to be displayed to the user, inject `org.jboss.seam.international.status.Messages` and call one of the Message factory methods. As mentioned earlier, factory methods accept either a plain-text template, or a `BundleKey`, specifying the name of the resource bundle to use, and the name of the key to use as a message template.

```
@Named
public class Example
{
  @Inject
  Messages messages;

  public String action()
  {
    messages.info("This is an {0} message, and will be displayed to {1}.", "INFO", "the user");
    return null;
  }
}
```

Adds the message: "This is an INFO message, and will be displayed to the user."

Notice how {0}, {1} ... {N} are replaced with the given parameters, and may be used more than once in a given template. In the case where a `BundleKey` is used to look up a message template, default text may be provided in case the resource cannot be loaded; default text uses the same parameters supplied for the bundle template. If no default text is supplied, a String representation of the `BundleKey` and its parameters will be displayed instead.

```
public String action()
{
  messages.warn(new BundleKey("org.jboss.seam.faces.exampleBundle", "messageKey"), "unique");
  return null;
```

```
}
```

`classpath:/org/jboss/seam/faces/exampleBundle.properties`

```
messageKey=This {0} parameter is not so {0}, see?
```

Adds the message: "This unique parameter is not so unique, see?"

3.2. Displaying pending messages

It's great when messages are added to the internal buffer, but it doesn't do much good unless the user actually sees them. In order to display messages, simply use the `<h:messages />` tag from JSF. Any pending messages will be displayed on the page just like normal `FacesMessages`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:s="http://jboss.org/seam/faces"
  xmlns:ui="http://java.sun.com/jsf/facelets">

  <h1>Welcome to Seam Faces!</h1>
  <p>All Messages and FacesMessages will be displayed below:</p>

  <h:messages />

</html>
```

Messages added to the internal buffer via the Messages API are stored in a central location during each request, and may be displayed by any view-technology that supports the Messages API. Seam Faces provides an integration that makes all of this automatic for you as a developer, and in addition, messages will automatically survive JSF navigation and redirects, as long as the redirect URL was encoded using `ExternalContext.encodeRedirectURL(...)`. If you are using JSF-compliant navigation, all of this is handled for you.

Seam Faces Components

While Seam Faces does not provide layout components or other UI-design related features, it does provide functional components designed to make developing JSF applications easier, more functional, more scalable, and more practical.

For layout and design components, take a look at *RichFaces* [http://jboss.org/richfaces], a UI component library specifically tailored for easy, rich web-interfaces.

4.1. Introduction

In order to use the Seam Faces components, you must first add the namespace to your view file, just like the standard JSF component libraries.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:s="http://jboss.org/seam/faces"
  xmlns:ui="http://java.sun.com/jsf/facelets">

  <h1>Welcome to Seam Faces!</h1>
  <p>
    This view imports the Seam Faces component library.
    Read on to discover what components it provides.
   </p>

</html>
```

Tip

All Seam Faces components use the following namespace: `http://jboss.org/seam/faces`

4.2. <s:validateForm>

On many occasions you might find yourself needing to compare the values of multiple input fields on a given page submit: confirming a password; re-enter password; address lookups; and so on. Performing cross-field form validation is simple - just place the <s:validateForm> component in the form you wish to validate, then attach your custom Validator.

```
<h:form id="locationForm">
```

```
  <h:inputText id="city" value="#{bean.city}" />
  <h:inputText id="state" value="#{bean.state}" />
  <h:inputText id="zip" value="#{bean.zip}" />
  <h:commandButton id="submit" value="Submit" action="#{bean.submitPost}" />

  <s:validateForm validatorId="locationValidator" />
</h:form>
```

The corresponding Validator for the example above would look something like this:

```
@FacesValidator("locationValidator")
public class LocationValidator implements Validator
{
  @Inject
  Directory directory;

  @Inject
  @InputField
  private Object city;

  @Inject
  @InputField
  private Object state;

  @Inject
  @InputField
  private ZipCode zip;

  @Override
  public void validate(final FacesContext context, final UIComponent comp, final Object values)
      throws ValidatorException
  {
    if(!directory.exists(city, state, zip))
    {
      throw new ValidatorException(
        new FacesMessage("Sorry, that location is not in our database. Please try again."));
    }
  }
}
```

Tip

You may inject the correct type directly.

```
@Inject
@InputField
private ZipCode zip;
```

Notice that the IDs of the inputText components match the IDs of your Validator @InputFields; each @Inject @InputField member will be injected with the value of the form input field who's ID matches the name of the variable.

In other words - the name of the @InputField annotated member variable will automatically be matched to the ID of the input component, unless overridden by using a field ID alias (see below.)

```
<h:form id="locationForm">
  <h:inputText id="cityId" value="#{bean.city}" />
  <h:inputText id="stateId" value="#{bean.state}" />
  <h:inputText id="zip" value="#{bean.zip}" />
  <h:commandButton id="submit" value="Submit" action="#{bean.submitPost}" />

  <s:validateForm fields="city=cityId state=stateId" validatorId="locationValidator" />
</h:form>
```

Notice that "zip" will still be referenced normally; you need only specify aliases for fields that differ in name from the Validator @InputFields.

Tip

`Using @InputField("customID")` with an ID override can also be used to specify a custom ID, instead of using the default: the name of the field. This gives you the ability to change the name of the private field, without worrying about changing the name of input fields in the View itself.

```
@Inject
@InputField("state")
private String sectorTwo;
```

An alternate way of accessing those fields on the validator by injecting an InputElement. It works similarly to @InputField, but stores the clientId and a JSF UIComponent, along with the field value.

```
@FacesValidator("fooValidator")
public class FooValidator implements Validator {
   @Inject
   private InputElement<String> firstNameElement;
   @Inject
   private InputElement<String> lastNameElement;

   @Inject
   private InputElement<Date> startDateElement;

   @Inject
   private InputElement<Date> endDateElement;
   ...

   }
```

Use get methods to access those information

```
public void validate(final FacesContext ctx, final UIComponent form, final Object value) throws ValidatorExceptio
      Date startDate = startDateElement.getValue();

      Calendar calendar = Calendar.getInstance();
      calendar.add(Calendar.DAY_OF_MONTH, -1);

      if (startDate.before(calendar.getTime())) {
         String message = messageBuilder.get().key(new DefaultBundleKey("booking_checkInNotFutureDate"))
               .targets(  startDateElement.getClientId()  ).build().getText();
         throw new ValidatorException(new FacesMessage(message));
      }
      ...
   }
```

4.3. <s:viewAction>

The view action component (`UIViewAction`) is an `ActionSource2` `UIComponent` that specifies an application-specific command (or action), using an EL method expression, to be invoked during one of the JSF lifecycle phases proceeding Render Response (i.e., view rendering).

View actions provide a lightweight front-controller for JSF, allowing the application to accommodate scenarios such as registration confirmation links, security and sanity checking a request (e.g., ensuring the resource can be loaded). They also allow JSF to work alongside action-oriented frameworks, and existing applications that use them.

4.3.1. Motivation

JSF employs an event-oriented architecture. Listeners are invoked in response to user-interface events, such as the user clicking on a button or changing the value of a form input. Unfortunately, the most important event on the web, a URL request (initiated by the user clicking on a link, entering a URL into the browser's location bar or selecting a bookmark), has long been overlooked in JSF. Historically, listeners have exclusively been activated on postback, which has led to the common complaint that in JSF, "everything is a POST."

We want to change that perception.

Processing a URL request event is commonly referred to as bookmarkable or GET support. Some GET support was added to JSF 2.0 with the introduction of view parameters and the pre-render view event. View parameters are used to bind query string parameters to model properties. The pre-render view event gives the developer a window to invoke a listener immediately prior to the view being rendered.

That's a start.

Seam brings the GET support full circle by introducing the view action component. A view action is the compliment of a `UICommand` for an initial (non-faces) request. Like its cohort, it gets executed by default during the Invoke Application phase (now used on both faces and non-faces requests). A view action can optionally be invoked on postback as well.

View actions (`UIViewAction`) are closely tied to view parameters (`UIViewParameter`). Most of the time, the view parameter is used to populate the model with data that is consumed by the method being invoked by a `UIViewAction` component, much like form inputs populate the model with data to support the method being invoked by a `UICommand` component.

4.3.2. Usage

Let's consider a typical scenario in web applications. You want to display the contents of a blog entry that matches the identifier specified in the URL. We'll assume the URL is:

```
http://localhost:8080/blog/entry.jsf?id=10
```

We'll use a view parameter to capture the identifier of the entry from the query string and a view action to fetch the entry from the database.

```
<f:metadata>
  <f:viewParam name="id" value="#{blogManager.entryId}"/>
```

```
  <s:viewAction action="#{blogManager.loadEntry}"/>
</f:metadata>
```

Tip

The view action component must be declared as a child of the view metadata facet (i.e., `<f:metadata>`) so that it gets incorporated into the JSF lifecycle on both non-faces (initial) requests and faces (postback) requests. If you put it anywhere else in the page, the behavior is undefined.

Warning

The JSF 2 specification specifies that there must be at least one view parameter for the view metadata facet to be processed on an initial request. This requirement was introduced into the JSF specification inadvertently. But not to worry. Seam Faces inserts a placeholder view parameter into the view metadata if it contains other components but no view parameters. That means you can use a view action without a view parameter, contrary to the JSF specification.

What do we do if the blog entry can't be found? View actions support declarative navigation just like `UICommand` components. So you can write a navigation rule that will be consulted before the page is rendered. If the rule matches, navigation occurs just as though this were a postback.

```
<navigation-rule>
    <from-view-id>/entry.xhtml</from-view-id>
    <navigation-case>
      <from-action>#{blogManager.loadEntry}</from-action>
      <if>#{empty entry}</if>
      <to-view-id>/home.xhtml</to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
```

After each view action is invoked, the navigation handler looks for a navigation case that matches the action's EL method signature and outcome. If a navigation case is matched, or the response is marked complete by the action, subsequent view actions are short-circuited. The lifecycle then advances appropriately.

By default, a view action is not executed on postback, since the primary intention of a view action is to support a non-faces request. If your application (or use case) is decidedly stateless, you

may need the view action to execute on any type of request. You can enable the view action on postback using the `onPostback` attribute:

```
<s:viewAction action="#{blogManager.loadEntry}" onPostback="true"/>
```

You may only want the view action to be invoked under certain conditions. For instance, you may only need it to be invoked if the conversation is transient. For that, you can use the `if` attribute, which accepts an EL value expression:

```
<s:viewAction action="#{blogEditor.loadEntry}" if="#{conversation.transient}"/>
```

There are two ways to control the phase in which the view action is invoked. You can set the `immediate` attribute to true, which moves the invocation to the Apply Request Values phase instead of the default, the Invoke Application phase.

```
<s:viewAction action="#{sessionManager.validateSession}" immediate="true"/>
```

You can also just specify the phase directly, using the name of the phase constant in the `PhaseId` class (the case does not matter).

```
<s:viewAction action="#{sessionManager.validateSession}" phase="APPLY_REQUEST_VALUES"/
>
```

Tip

The valid phases for a view action are:

- `APPLY_REQUEST_VALUES` (default if `immediate="true"`)
- `UPDATE_MODEL_VALUES`
- `PROCESS_VALIDATIONS`
- `INVOKE_APPLICATION` (default)

If the phase is set, it takes precedence over the immediate flag.

4.3.3. View actions vs the PreRenderViewEvent

The purpose of the view action is similar to use of the PreRenderViewEvent. In fact, the code to load a blog entry before the page is rendered could be written as:

```
<f:metadata>
  <f:viewParam name="id" value="#{blogManager.entryId}"/>
  <f:event type="preRenderView" listener="#{blogManager.loadEntry}"/>
</f:metadata>
```

However, the view action has several important advantages:

- It's lightweight
- It's timing can be controlled
- It's contextual
- It can trigger navigation

View actions are lightweight because they get processed on a non-faces (initial) request *before* the full component tree is built. When the view actions are invoked, the component tree only contains view metadata.

As demonstrated above, you can specify a prerequisite condition for invoking the view action, control whether it's invoked on postback, specify the phase in which it's invoked and tie the invocation into the declarative navigation system. The PreRenderViewEvent is quite basic in comparison.

4.4. UI Input Container

UIInputContainer is a supplemental component for a JSF 2.0 composite component encapsulating one or more input components (EditableValueHolder), their corresponding message components (UIMessage) and a label (HtmlOutputLabel).

This component takes care of wiring the label to the first input and the messages to each input in sequence. It also assigns two implicit attribute values, "required" and "invalid" to indicate that a required input field is present and whether there are any validation errors, respectively. To determine if a input field is required, both the required attribute is consulted and whether the property has Bean Validation constraints.

Finally, if the "label" attribute is not provided on the composite component, the label value will be derived from the id of the composite component, for convenience.

There's a composite componente that ships with seam-faces under the url:

http://java.sun.com/jsf/composite/components/seamfaces.

```
xmlns:sc="http://java.sun.com/jsf/composite/components/seamfaces"
   <sc:inputContainer label="name" id="name">
      <h:inputText id="input" value="#{person.name}"/>
   </sc:inputContainer>
```

If you want to define your own composite component, follow this definition example (minus layout):

```
<cc:interface componentType="org.jboss.seam.faces.InputContainer"/>
 <cc:implementation>
   <h:outputLabel id="label" value="#{cc.attrs.label}:" styleClass="#{cc.attrs.invalid ? 'invalid' :
 ''}">
    <h:outputText styleClass="required" rendered="#{cc.attrs.required}" value="*"/>
  </h:outputLabel>
  <h:panelGroup>
    <cc:insertChildren/>
  </h:panelGroup>
  <h:message id="message" errorClass="invalid message" rendered="#{cc.attrs.invalid}"/>
 </cc:implementation>
```

Tip

it's currently required to wrap the insertChildren tag with a jsf panelGroup. Please see http://java.net/jira/browse/JAVASERVERFACES-1991 for more details.

Tip

NOTE: Firefox does not properly associate a label with the target input if the input id contains a colon (:), the default separator character in JSF. JSF 2 allows developers to set the value via an initialization parameter (context-param in web.xml) keyed to `javax.faces.SEPARATOR_CHAR`. We recommend that you override this setting to make the separator an underscore (_).

Faces Artifact Injection

One of the goals of the Seam Faces Module is to make support for CDI a more ubiquitous experience, by allowing injection of JSF Lifecycle Artifacts into managed beans, and also by providing support for @Inject where it would not normally be available. This section describes the additional CDI integration for faces artifact injection

5.1. @*Scoped and @Inject in Validators and Converters

Frequently when performing complex validation, it is necessary to access data stored in a database or in other contextual objects within the application itself. JSF does not, by default, provide support for `@Inject` in Converters and Validators, but Seam Faces makes this available. In addition to injection, it is sometimes convenient to be able to scope a validator just as we would scope a managed bean; this feature is also added by Seam Faces.

Notice how the Validator below is actually `@RequestScoped`, in addition to using injection to obtain an instance of the `UserService` with which to perform an email database lookup.

```
@RequestScoped
@FacesValidator("emailAvailabilityValidator")
public class EmailAvailabilityValidator implements Validator
{
   @Inject
   UserService us;

   @Override
   public void validate(final FacesContext context, final UIComponent component, final Object value)
         throws ValidatorException
   {
      String field = value.toString();
      try
      {
         us.getUserByEmail(field);
         FacesMessage msg = new FacesMessage("That email address is unavailable");
         throw new ValidatorException(msg);
      }
      catch (NoSuchObjectException e)
      {
      }
   }
}
```

Warning

We recommend to always use `@RequestScoped` converters/validators unless a longer scope is required, in which case you should use the appropriate scope annotation, but it should not be omitted.

Because of the way JSF persists Validators between requests, particularly when using `@Inject` inside a validator or converter, forgetting to use a `@*Scoped` annotation could in fact cause `@Inject`'ed objects to become null.

An example Converter using @Inject

```
@SessionScoped
@FacesConverter("authorConverter")
public class UserConverter implements Converter
{
  @Inject
  private UserService service;

  @PostConstruct
  public void setup()
  {
    System.out.println("UserConverter started up");
  }

  @PreDestroy
  public void shutdown()
  {
    System.out.println("UserConverter shutting down");
  }

  @Override
  public Object getAsObject(final FacesContext arg0, final UIComponent arg1, final String userName)
  {
    // ...
    return service.getUserByName(userName);
  }

  @Override
  public String getAsString(final FacesContext context, final UIComponent comp, final Object user)
  {
    // ...
    return ((User)user).getUsername();
```

```
  }
}
```

5.2. @Inject'able Faces Artifacts

This is the list of inject-able artifacts provided through Seam Faces. These objects would normally require static method-calls in order to obtain handles, but Seam Faces attempts to break that coupling by providing @Inject'able artifacts. This means it will be possible to more easily provide mocked objects during unit and integration testing, and also simplify bean code in the application itself.

Artifact Class	Example

javax.faces.context.FacesContext	`public class Bean {`
	`  @Inject FacesContext context;`
	`}`

javax.faces.context.ExternalContext	`public class Bean {`
	`  @Inject ExternalContext context;`
	`}`

javax.faces.application.NavigationH	`public class Bean {`
	`  @Inject NavigationHandler handler;`
	`}`

javax.faces.context.Flash	`public class Bean {`
	`  @Inject Flash flash;`
	`}`

Faces Events Propagation

When the seam-faces module is installed in a web application, JSF events will automatically be propagated via the CDI event-bridge, enabling managed beans to easily observe all Faces events.

There are two categories of events: JSF phase events, and JSF system events. Phase events are triggered as JSF processes each lifecycle phase, while system events are raised at more specific, fine-grained events during request processing.

6.1. JSF Phase events

A JSF phase listener is a class that implements `javax.faces.event.PhaseListener` and is registered in the web application's `faces-config.xml` file. By implementing the methods of the interfaces, the user can observe events fired before or after any of the six lifecycle phases of a JSF request: `restore view`, `apply request values`, `process validations`, `update model values`, `invoke application` or `render view`.

Tip

In order to observe events in an EJB JAR, the beans.xml file must be in both the WEB-INF folder of the WAR, and inside the EJB JAR containing the observer.

6.1.1. Seam Faces Phase events

What Seam provides is propagation of these Phase events to the CDI event bus; therefore, you can observe events using normal CDI `@Observes` methods. Bringing the events to CDI beans removes the need to register phase listener classes via XML, and gives the added benefit of injection, alternatives, interceptors and access to all other features of CDI.

Creating an observer method in CDI is simple; just provide a method in a managed bean that is annotated with `@Observes`. Each observer method must accept at least one method parameter: the event object; the type of this object determines the type of event being observed. Additional parameters may also be specified, and their values will be automatically injected by the container as per the CDI specification.

In this case, the event object passed along from the phase listener is a `javax.faces.event.PhaseEvent`. The following example observes all Phase events.

```
public void observeAll(@Observes PhaseEvent e)
{
   // Do something with the event object
}
```

Events can be further filtered by adding Qualifiers. The name of the method itself is not significant. (See the CDI Reference Guide for more information on events and observing.)

Since the example above simply processes all events, however, it might be appropriate to filter out some events that we aren't interested in. As stated earlier, there are six phases in the JSF lifecycle, and an event is fired before and after each, for a total of 12 events. The `@Before` and `@After` "temporal" qualifiers can be used to observe events occurring only before or only after a Phase event. For example:

```
public void observeBefore(@Observes @Before PhaseEvent e)
{
   // Do something with the "before" event object
}

public void observeAfter(@Observes @After PhaseEvent e)
{
   // Do something with the "after" event object
}
```

If we are interested in both the "before" and "after" event of a particular phase, we can limit them by adding a "lifecycle" qualifier that corresponds to the phase:

```
public void observeRenderResponse(@Observes @RenderResponse PhaseEvent e)
{
   // Do something with the "render response" event object
}
```

By combining a temporal and lifecycle qualifier, we can achieve the most specific qualification:

```
public void observeBeforeRenderResponse(@Observes @Before @RenderResponse PhaseEvent e)
{
   // Do something with the "before render response" event object
}
```

6.1.2. Phase events listing

This is the full list of temporal and lifecycle qualifiers

Qualifier	Type	Description
@Before	temporal	Qualifies events before lifecycle phases
@After	temporal	Qualifies events after lifecycle phases
@RestoreView	lifecycle	Qualifies events from the "restore view" phase
@ApplyRequestValues	lifecycle	Qualifies events from the "apply request values" phase
@ProcessValidations	lifecycle	Qualifies events from the "process validations" phase
@UpdateModelValues	lifecycle	Qualifies events from the "update model values" phase
@InvokeApplication	lifecycle	Qualifies events from the "invoke application" phase
@RenderResponse	lifecycle	Qualifies events from the "render response" phase

The event object is always a `javax.faces.event.PhaseEvent` and according to the general CDI principle, filtering is tightened by adding qualifiers and loosened by omitting them.

6.2. JSF system events

Similar to JSF Phase Events, System Events take place when specific events occur within the JSF life-cycle. Seam Faces provides a bridge for all JSF System Events, and propagates these events to CDI.

6.2.1. Seam Faces System events

This is an example of observing a Faces system event:

```
public void observesThisEvent(@Observes ExceptionQueuedEvent e)
{
   // Do something with the event object
}
```

6.2.2. System events listing

Since all JSF system event objects are distinct, no qualifiers are needed to observe them. The following events may be observed:

Event object	Context	Description
SystemEvent	all	All events
ComponentSystemEvent	component	All component events
PostAddToViewEvent	component	After a component was added to the view
PostConstructViewMapEvent	component	After a view map was created
PostRestoreStateEvent	component	After a component has its state restored

Event object	Context	Description
PostValidateEvent	component	After a component has been validated
PreDestroyViewMapEvent	component	Before a view map has been restored
PreRemoveFromViewEvent	component	Before a component has been removed from the view
PreRenderComponentEvent	component	After a component has been rendered
PreRenderViewEvent	component	Before a view has been rendered
PreValidateEvent	component	Before a component has been validated
ExceptionQueuedEvent	system	When an exception has been queued
PostConstructApplicationEvent	system	After the application has been constructed
PostConstructCustomScopeEvent	system	After a custom scope has been constructed
PreDestroyApplicationEvent	system	Before the application is destroyed
PreDestroyCustomScopeEvent	system	Before a custom scope is destroyed

6.2.3. Component system events

There is one qualifier, `@Component` that can be used with component events by specifying the component ID. Note that view-centric component events `PreRenderViewEvent`, `PostConstructViewMapEvent` and `PreDestroyViewMapEvent` do not fire with the `@Component` qualifier.

```
public void observePrePasswordValidation(@Observes @Component("form:password") PreValidateEvent e)
{
// Do something with the "before password is validated" event object
}
```

Global system events are observer without the component qualifier

```
public void observeApplicationConstructed(@Observes PostConstructApplicationEvent e)
{
// Do something with the "after application is constructed" event object
}
```

The name of the observing method is not relevant; observers are defined solely via annotations.

Faces View Configuration

Seam Faces aims to provide JSF web developers with a truly worthy framework for web development by ironing out some of the JSF pain points, integrating tightly with CDI, and offering out of the box integration with the other Seam Modules. The view configuration presented here provides a central means of configuring seemingly disparate concerns.

Adhering to the CDI core tenet of type safety, Seam Faces offers a type-safe way to configure the behaviour of your JSF views. So far these configurable behaviors include:

- Configuring view access by integrating with Seam Security
- Configuring URL rewriting by integrating with Pretty Faces (or any other pluggable URL rewriting framework)
- Configuring Transactional behaviour through Seam Persistence
- A personal favorite: setting faces-direct=true when navigating to a view.

The Seam Faces example application `faces-viewconfig`, demonstrates all the view configuration techniques discussed in this chapter.

7.1. Configuration With Annotated Enums

Faces pages are configured by placing annotations on the properties of an Java `enum`. The annotation `@ViewConfig` is placed on a Java `interface`, which contains a static enum. It is the properties of this static enum that hold the individual annotations used to configure the view.

```
@ViewConfig
public interface Pages {

   static enum Pages1 {

      @ViewPattern("/admin.xhtml")
      @Admin
      ADMIN,

      @UrlMapping(pattern="/item/#{item}/")
      @ViewPattern("/item.xhtml")
      @Owner
      ITEM,
```

```
        @FacesRedirect
        @ViewPattern("/*")
        @AccessDeniedView("/denied.xhtml")
        @LoginView("/login.xhtml")
        ALL;

    }
}
```

The `interface` containing the enum is required due to a limitation in version 1.0 of the CDI specification. If the `@ViewConfig` is placed directly on the enum, the CDI specification does not require the annotations to be scanned.

Each property of the enum is annotated with at least the `@ViewPattern` annotation. This view pattern is used to match against the JSF view ids, to determine which annotations apply to a given view id. The view patterns themselves suport the `*` wild character. A view is matched against all view parameters that apply to determine the relevant annotations. If conflicting annotatins are found, the annotation paired with the most specific matching view pattern takes precedence.

7.2. Configuring View Restrictions

Securing view access is achieved through integration with Seam Security, which must be explilcitly bundled with your web application. Refer to the Seam Security documentation for details on how to setup authentication. What we'll cover here is the authorisation to JSF views.

7.2.1. Writing Seam security Annotations

To secure a view, we start by writing an annotation qualified with a `@SecurityBindingType` qualifier:

```
@SecurityBindingType
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
public @interface Owner {
}
```

This `@SecurityBindingType` qualified annotation is placed on an enum property in the `@ViewConfig` annotated `interface`, adjacent to the @ViewPattern to which the security restriction applies. View patterns with wildcards are supported for security based annotations.

7.2.2. Applying the Security Restrictions

Methods that enforce the Security restriction are annotated with the `@Secures` annotation, as well as the same `@SecurityBindingType` qualified annoation used on the `@ViewConfig enum` property.

```
public @Secures @Owner boolean ownerChecker(Identity identity, @Current Item item) {
      if (item == null || identity.getUser() == null) {
         return false;
      } else {
         return item.getOwner().equals(identity.getUser().getId());
      }
   }
```

When a JSF view is visited, matching `@ViewPattern` patterns are found, and their associated `@SecurityBindingType` qualified annoations are looked up. The corresponding method is invoked, and access is either granted or denied. If access is denied, and the user is not yet logged in, the user will be redirected to a view specified in a `@LoginView` annotation for that view. However if access is denied, and the user is logged in, navigation will be redirected to a view specified in the `@AccessDeined` annotation.

Refer to the Seam Security documentation for further details on writing `@Secures` methods for restricting view access, including supoort for parameter injection.

7.2.3. Changing the Restrcition Phases

By default, Security restrictions are enforced before the `Invoke Application` phase, and before the `Render Response` phase. Restrictions are enforced twice per lifecylce, as the view id normally changes at the end of the `Invoke Application` phase. However, use cases exist requiring enforcement of a Security restriction at a different phase. For instance it is desirable to enforce a role-based restriction as early in the lifecycle as possible, to prevent any unnecessary computations from occurring. This is achieved using the `@RestrictAtView` annotation.

By qualying a `@SecurityBindingType` qualified annoation with the `@RestrictAtView` qualifier, one is able to specify at which phase that individual Security restriction should be applied. Additionally, the `@RestrictAtView` annotation can be applied directly to a `@ViewConfig enum` property, to determine the restriction phase of all associated `@SecurityBindingType` qualified annoations.

7.3. Configuring URL Rewriting

Seam Faces delegates URL Rewriting to *Pretty Faces* [http://ocpsoft.com/prettyfaces/]. The Rewriting mechanism however is pluggable, and an alternate URL Rewriting engine could easily

be used instead. What makes SeamFaces unique in it's approach to URL rewriting, is that the rewrite configuration is done via the `@ViewConfig` mechanism, so all view configuration is done consistently.

To configure UrlRewriting, use the `@UrlRewrite` Seam Faces annotation:

```
...
@UrlMapping(pattern="/item/#{item}/")
@ViewPattern("/item.xhtml")
ITEM;
...
```

The above listing would rewrite the url `/item.jsf/item=2` into `/item/2/`. See the Pretty Faces documentation for further details on configuring URL rewriting.

7.4. Configuring "faces-redirect"

A `@ViewPattern` annotated with `@FacesRedirect` will cause all navigations to views that match that pattern to have their faces-redirect property set to true. This alleviates the requirement to append `?faces-redirect=true` to all implicit navigation rules, and neither does it have to be specified in the navigation rules defined in the faces-config.xml.

7.5. Configuring Transactional Views

Through integration with Seam Persistence, the transactional behaviour of the JSF lifecycle can be configured through the `@ViewConfig` mechanism. More details will be provided here, as this feature is fleshed out.