Seam Servlet Module

Reference Guide

Dan Allen

Lincoln Baxter III

Nicklas Karlsson

Introduction

The goal of the Seam Servlet module is to provide portable enhancements to the Servlet API. Features include producers for implicit Servlet objects and HTTP request state, propagating Servlet events to the CDI event bus, forwarding uncaught exceptions to the Seam Catch handler chain and binding the BeanManager to a Servlet context attribute for convenient access.

Installation

To use the Seam Servlet module, you need to put the API and implementation JARs on the classpath of your web application. Most of the features of Seam Servlet 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 Servlet:

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

Tip

Substitute the expression ${seam.servlet.version} with the most recent or appropriate version of Seam Servlet. 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.servlet</groupId>
   <artifactId>seam-servlet-api</artifactId>
   <version>${seam.servlet.version}</version>
   <scope>compile</scope>
</dependency>

<dependency>
   <groupId>org.jboss.seam.servlet</groupId>
   <artifactId>seam-servlet-impl</artifactId>
   <version>${seam.servlet.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.servlet.event.ServletEventBridgeListener</listener-class>
</listener>

<servlet>
   <servlet-name>Servlet Event Bridge Servlet</servlet-name>
   <servlet-class>org.jboss.seam.servlet.event.ServletEventBridgeServlet</servlet-class>
   <!-- Make load-on-startup large enough to be initialized last (thus destroyed first) -->
   <load-on-startup>99999</load-on-startup>
</servlet>

<filter>
   <filter-name>Catch Exception Filter</filter-name>
   <filter-class>org.jboss.seam.servlet.CatchExceptionFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>Catch Exception Filter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter>
   <filter-name>Servlet Event Bridge Filter</filter-name>
   <filter-class>org.jboss.seam.servlet.event.ServletEventBridgeFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>Servlet Event Bridge Filter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Warning

In order for the Seam Servlet event bridge to properly fire the ServletContext initialized event, the CDI runtime must be started at the time the Seam Servlet listener is invoked. This ordering is guaranteed in a compliant Java EE 6 environment. If you are using a CDI implementation in a Servlet environment (e.g., Weld Servlet), and it relies on a Servlet listener to bootstrap, that listener must be registered *before* any Seam Servlet listener in web.xml.

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

Servlet event propagation

By including the Seam Servlet module in your web application (and performing the necessary *listener configuration* for pre-Servlet 3.0 environments), the servlet lifecycle events will be propagated to the CDI event bus so you can observe them using observer methods on CDI beans. Seam Servlet also fires additional lifecycle events not offered by the Servlet API, such as when the response is initialized and destroyed.

2.1. Servlet context lifecycle events

This category of events corresponds to the event receivers on the `javax.servlet.ServletContextListener` interface. The event propagated is a `javax.servlet.ServletContext` (not a `javax.servlet.ServletContextEvent`, since the `ServletContext` is the only relevant information this event provides).

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@Initialized` and `@Destroyed`) that can be used to observe a specific lifecycle phase of the servlet context.

The servlet context lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.ServletContext	The servlet context is initialized or destroyed
@Initialized	javax.servlet.ServletContext	The servlet context is initialized
@Destroyed	javax.servlet.ServletContext	The servlet context is destroyed

If you want to listen to both lifecycle events, leave out the qualifiers on the observer method:

```
public void observeServletContext(@Observes ServletContext ctx) {
   System.out.println(ctx.getServletContextName() + " initialized or destroyed");
}
```

If you are interested in only a particular lifecycle phase, use one of the provided qualifers:

```
public void observeServletContextInitialized(@Observes @Initialized ServletContext ctx) {
   System.out.println(ctx.getServletContextName() + " initialized");
}
```

As with all CDI observers, the name of the method is insignificant.

These events are fired using a built-in servlet context listener. The CDI environment will be active when these events are fired (including when Weld is used in a Servlet container). The listener is

configured to come before listeners in other extensions, so the initialized event is fired before other servlet context listeners are notified and the destroyed event is fired after other servlet context listeners are notified. However, this order cannot be not guaranteed if another extension library is also configured to be ordered before others.

2.2. Application initialization

The servlet context initialized event described in the previous section provides an ideal opportunity to perform startup logic *as an alterative to using an EJB 3.1 startup singleton*. Even better, you can configure the bean to be destroyed immediately following the initialization routine by leaving it as dependent scoped (dependent-scoped observers only live for the duration of the observe method invocation).

Here's an example of entering seed data into the database in a development environment (as indicated by a stereotype annotation named `@Development`).

```
@Stateless
@Development
public class SeedDataImporter {
   @PersistenceContext
   private EntityManager em;

   public void loadData(@Observes @Initialized ServletContext ctx) {
      em.persist(new Product(1, "Black Hole", 100.0));
   }
}
```

If you'd rather not tie yourself to the Servlet API, you can observe the `org.jboss.seam.servlet.WebApplication` rather than the `ServletContext`. `WebApplication` is a informational object provided by Seam Servlet that holds select information about the `ServletContext` such as the application name, context path, server info and start time.

The web application lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	WebApplication	The web application is initialized, started or destroyed
@Initialized	WebApplication	The web application is initialized
@Started	WebApplication	The web application is started (ready)
@Destroyed	WebApplication	The web application is destroyed

Here's the equivalent of receiving the servlet context initialized event without coupling to the Servlet API:

```
public void loadData(@Observes @Initialized WebApplication webapp) {
    System.out.println(webapp.getName() + " initialized at " + new Date(webapp.getStartTime()));
}
```

If you want to perform initialization as late as possible, after all other initialization of the application is complete, you can observe the `WebApplication` event qualified with `@Started`.

```
public void onStartup(@Observes @Started WebApplication webapp) {
    System.out.println("Application at " + webapp.getContextPath() + " ready to handle requests");
}
```

The `@Started` event is fired in the init method of a built-in Servlet with a load-on-startup value of 99999.

You can also use `WebApplication` with the `@Destroyed` qualifier to be notified when the web application is stopped. This event is fired by the aforementioned built-in Servlet during it's destroy method, so likely it should fire when the application is first released.

```
public void onShutdown(@Observes @Destroyed WebApplication webapp) {
        System.out.println("Application at " + webapp.getContextPath() + " no longer handling requests");
}
```

2.3. Servlet request lifecycle events

This category of events corresponds to the event receivers on the `javax.servlet.ServletRequestListener` interface. The event propagated is a `javax.servlet.ServletRequest` (not a `javax.servlet.ServletRequestEvent`, since the `ServletRequest` is the only relevant information this event provides).

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@Initialized` and `@Destroyed`) that can be used to observe a specific lifecycle phase of the servlet request and a secondary qualifier to filter events by servlet path (`@Path`).

The servlet request lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.ServletRequest	A servlet request is initialized or destroyed
@Initialized	javax.servlet.ServletRequest	A servlet request is initialized
@Destroyed	javax.servlet.ServletRequest	A servlet request is destroyed

Qualifier	Type	Description
@Default (optional)	javax.servlet.http.HttpServletRequest	Any HTTP servlet request is initialized or destroyed
@Initialized	javax.servlet.http.HttpServletRequest	Any HTTP servlet request is initialized
@Destroyed	javax.servlet.http.HttpServletRequest	Any HTTP servlet request is destroyed
@Path(PATH)	javax.servlet.http.HttpServletRequest	Selects HTTP request with servlet path matching PATH (drop leading slash)

If you want to listen to both lifecycle events, leave out the qualifiers on the observer:

```
public void observeRequest(@Observes ServletRequest request) {
    // Do something with the servlet "request" object
}
```

If you are interested in only a particular lifecycle phase, use a qualifer:

```
public void observeRequestInitialized(@Observes @Initialized ServletRequest request) {
    // Do something with the servlet "request" object upon initialization
}
```

You can also listen specifically for a `javax.servlet.http.HttpServletRequest` simply by changing the expected event type.

```
public void observeRequestInitialized(@Observes @Initialized HttpServletRequest request) {
    // Do something with the HTTP servlet "request" object upon initialization
}
```

You can associate an observer with a particular servlet request path (exact match, no leading slash).

```
public void observeRequestInitialized(@Observes @Initialized @Path("offer") HttpServletRequest request) {
    // Do something with the HTTP servlet "request" object upon initialization
    // only when servlet path /offer is requested
}
```

As with all CDI observers, the name of the method is insignificant.

These events are fired using a built-in servlet request listener. The listener is configured to come before listeners in other extensions, so the initialized event is fired before other servlet request listeners are notified and the destroyed event is fired after other servlet request listeners are notified. However, this order cannot be not guaranteed if another extension library is also configured to be ordered before others.

2.4. Servlet response lifecycle events

The Servlet API does not provide a listener for accessing the lifecycle of a response. Therefore, Seam Servlet simulates a response lifecycle listener using CDI events. The event object fired is a `javax.servlet.ServletResponse`.

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@Initialized` and `@Destroyed`) that can be used to observe a specific lifecycle phase of the servlet response and a secondary qualifier to filter events by servlet path (`@Path`).

The servlet response lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.ServletResponse	A servlet response is initialized or destroyed
@Initialized	javax.servlet.ServletResponse	A servlet response is initialized
@Destroyed	javax.servlet.ServletResponse	A servlet response is destroyed
@Default (optional)	javax.servlet.http.HttpServletResponse	A servlet response is initialized or destroyed
@Initialized	javax.servlet.http.HttpServletResponse	A servlet response is initialized
@Destroyed	javax.servlet.http.HttpServletResponse	A servlet response is destroyed
@Path(PATH)	javax.servlet.http.HttpServletResponse	Selects HTTP response with servlet path matching PATH (drop leading slash)

If you want to listen to both lifecycle events, leave out the qualifiers.

```
public void observeResponse(@Observes ServletResponse response) {
   // Do something with the servlet "response" object
}
```

If you are interested in only a particular one, use a qualifer

```
public void observeResponseInitialized(@Observes @Initialized ServletResponse response) {
   // Do something with the servlet "response" object upon initialization
}
```

You can also listen specifically for a `javax.servlet.http.HttpServletResponse` simply by changing the expected event type.

```
public void observeResponseInitialized(@Observes @Initialized HttpServletResponse response) {
   // Do something with the HTTP servlet "response" object upon initialization
}
```

If you need access to the `ServletRequest` and/or the `ServletContext` objects at the same time, you can simply add them as parameters to the observer methods. For instance, let's assume you want to manually set the character encoding of the request and response.

```
public void setupEncoding(@Observes @Initialized ServletResponse res, ServletRequest req) throws Exception {
   if (this.override || req.getCharacterEncoding() == null) {
      req.setCharacterEncoding(encoding);
      if (override) {
         res.setCharacterEncoding(encoding);
      }
   }
}
```

As with all CDI observers, the name of the method is insignificant.

Tip

If the response is committed by one of the observers, the request will not be sent to the target Servlet and the filter chain is skipped.

2.5. Servlet request context lifecycle events

Rather than having to observe the request and response as separate events, or include the request object as an parameter on a response observer, it would be convenient to be able to observe them as a pair. That's why Seam Servlet fires an synthetic lifecycle event for the wrapper type `ServletRequestContext`. The `ServletRequestContext` holds the `ServletRequest` and the `ServletResponse` objects, and also provides access to the `ServletContext`.

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@Initialized` and `@Destroyed`) that can be used to observe a specific lifecycle phase of the servlet request context and a secondary qualifier to filter events by servlet path (`@Path`).

The servlet request context lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	ServletRequestContext	A request is initialized or destroyed
@Initialized	ServletRequestContext	A request is initialized
@Destroyed	ServletRequestContext	A request is destroyed
@Default (optional)	HttpServletRequestContext	An HTTP request is initialized or destroyed
@Initialized	HttpServletRequestContext	An HTTP request is initialized
@Destroyed	HttpServletRequestContext	An HTTP request is destroyed
@Path(PATH)	HttpServletRequestContext	Selects HTTP request with servlet path matching PATH (drop leading slash)

Let's revisit the character encoding observer and examine how it can be simplified by this event:

```
public void setupEncoding(@Observes @Initialized ServletRequestContext ctx) throws Exception {
   if (this.override || ctx.getRequest().getCharacterEncoding() == null) {
      ctx.getRequest().setCharacterEncoding(encoding);
      if (override) {
         ctx.getResponse().setCharacterEncoding(encoding);
      }
   }
}
```

You can also observe the `HttpServletRequestContext` to be notified only on HTTP requests.

Tip

If the response is committed by one of the observers, the request will not be sent to the target Servlet and the filter chain is skipped.

Since observers that have access to the response can commit it, an `HttpServletRequestContext` observer that receives the initialized event can effectively work as a filter or even a Servlet. Let's consider a primitive welcome page filter that redirects visitors to the start page:

```
public void redirectToStartPage(@Observes @Path("") @Initialized HttpServletRequestContext ctx)
      throws Exception {
   String startPage = ctx.getResponse().encodeRedirectURL(ctx.getContextPath() + "/start.jsf");
   ctx.getResponse().sendRedirect(startPage);
}
```

Now you never have to write a Servlet listener, Servlet or Filter again!

2.6. Session lifecycle events

This category of events corresponds to the event receivers on the `javax.servlet.http.HttpSessionListener` interface. The event propagated is a `javax.servlet.http.HttpSession` (not a `javax.servlet.http.HttpSessionEvent`, since the `HttpSession` is the only relevant information this event provides).

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@Initialized` and `@Destroyed`) that can be used to observe a specific lifecycle phase of the session.

The session lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.http.HttpSession	The session is initialized or destroyed
@Initialized	javax.servlet.http.HttpSession	The session is initialized
@Destroyed	javax.servlet.http.HttpSession	The session is destroyed

If you want to listen to both lifecycle events, leave out the qualifiers. Note that omitting all qualifiers will observe all events with a `HttpSession` as event object.

```
public void observeSession(@Observes HttpSession session) {
   // Do something with the "session" object
}
```

If you are interested in only a particular one, use a qualifer

```
public void observeSessionInitialized(@Observes @Initialized HttpSession session) {
   // Do something with the "session" object upon being initialized
}
```

As with all CDI observers, the name of the method is insignificant.

2.7. Session activation events

This category of events corresponds to the event receivers on the `javax.servlet.http.HttpSessionActivationListener` interface. The event propagated is a `javax.servlet.http.HttpSession` (not a `javax.servlet.http.HttpSessionEvent`, since the `HttpSession` is the only relevant information this event provides).

There are two qualifiers provided in the `org.jboss.seam.servlet.event` package (`@DidActivate` and `@WillPassivate`) that can be used to observe a specific lifecycle phase of the session.

The session activation events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.http.HttpSession	The session is initialized or destroyed
@DidActivate	javax.servlet.http.HttpSession	The session is activated
@WillPassivate	javax.servlet.http.HttpSession	The session will passivate

If you want to listen to both lifecycle events, leave out the qualifiers. Note that omitting all qualifiers will observe all events with a `HttpSession` as event object.

```
public void observeSession(@Observes HttpSession session) {
   // Do something with the "session" object
}
```

If you are interested in only a particular one, use a qualifer

```
public void observeSessionCreated(@Observes @WillPassivate HttpSession session) {
   // Do something with the "session" object when it's being passivated
}
```

As with all CDI observers, the name of the method is insignificant.

Injectable Servlet objects and request state

Seam Servlet provides producers that expose a wide-range of information available in a Servlet environment (e.g., implicit objects such as ServletContext and HttpSession and state such as HTTP request parameters) as beans. You access this information by injecting the beans produced. This chapter documents the Servlet objects and request state that Seam Servlet exposes and how to inject them.

3.1. @Inject @RequestParam

The `@RequestParam` qualifier allows you to inject an HTTP request parameter (i.e., URI query string or URL form encoded parameter).

Assume a request URL of /book.jsp?id=1.

```
@Inject @RequestParam("id")
private String bookId;
```

The value of the specified request parameter is retrieved using the method `ServletRequest.getParameter(String)`. It is then produced as a dependent-scoped bean of type String qualified `@RequestParam`.

The name of the request parameter to lookup is either the value of the `@RequestParam` annotation or, if the annotation value is empty, the name of the injection point (e.g., the field name).

Here's the example from above modified so that the request parameter name is implied from the field name:

```
@Inject @RequestParam
private String id;
```

If the request parameter is not present, and the injection point is annotated with `@DefaultValue`, the value of the `@DefaultValue` annotation is returned instead.

Here's an example that provides a fall-back value:

```
@Inject @RequestParam @DefaultValue("25")
private String pageSize;
```

If the request parameter is not present, and the `@DefaultValue` annotation is not present, a null value is injected.

Warning

Since the bean produced is dependent-scoped, use of the `@RequestParam` annotation on class fields and bean properties is only safe for request-scoped beans. Beans with wider scopes should wrap this bean in an `Instance` bean and retrieve the value within context of the thread in which it's needed.

```
@Inject @RequestParam("id")
private Instance<String> bookIdResolver;
...
String bookId = bookIdResolver.get();
```

3.2. @Inject @HeaderParam

Similar to the `@RequestParam`, you can use the `@HeaderParam` qualifier to inject an HTTP header parameter. Here's an example of how you inject the user agent string of the client that issued the request:

```
@Inject @HeaderParam("User-Agent")
private String userAgent;
```

The `@HeaderParam` also supports a default value using the `@DefaultValue` annotation.

Warning

Since the bean produced is dependent-scoped, use of the `@HeaderParam` annotation on class fields and bean properties is only safe for request-scoped beans. Beans with wider scopes should wrap this bean in an `Instance` bean and retrieve the value within context of the thread in which it's needed.

```
@Inject @HeaderParam("User-Agent")
private Instance<String> userAgentResolver;
...
String userAgent = userAgentResolver.get();
```

3.3. @Inject ServletContext

The `ServletContext` is made available as an application-scoped bean. It can be injected safetly into any CDI bean as follows:

```
@Inject
private ServletContext context;
```

The producer obtains a reference to the `ServletContext` by observing the `@Initialized ServletContext` event raised by this module's Servlet-to-CDI event bridge.

3.4. @Inject ServletRequest / HttpServletRequest

The `ServletRequest` is made available as a request-scoped bean. If the current request is an HTTP request, the produced bean is an `HttpServletRequest`. It can be injected safetly into any CDI bean as follows:

```
@Inject
private ServletRequest request;
```

or, for HTTP requests

```
@Inject
private HttpServletRequest httpRequest;
```

The producer obtains a reference to the `ServletRequest` by observing the `@Initialized ServletRequest` event raised by this module's Servlet-to-CDI event bridge.

3.5. @Inject ServletResponse / HttpServletResponse

The `ServletResponse` is made available as a request-scoped bean. If the current request is an HTTP request, the produced bean is an `HttpServletResponse`. It can be injected safetly into any CDI bean as follows:

```
@Inject
private ServletResponse reponse;
```

or, for HTTP requests

```
@Inject
private HttpServletResponse httpResponse;
```

The producer obtains a reference to the `ServletResponse` by observing the `@Initialized ServletResponse` event raised by this module's Servlet-to-CDI event bridge.

3.6. @Inject HttpSession

The `HttpSession` is made available as a request-scoped bean. It can be injected safetly into any CDI bean as follows:

```
@Inject
private HttpSession session;
```

Injecting the `HttpSession` will force the session to be created. The producer obtains a reference to the `HttpSession` by calling the `getSession()` on the `HttpServletRequest`. The reference to the `HttpServletRequest` is obtained by observing the `@Initialized HttpServletRequest` event raised by this module's Servlet-to-CDI event bridge.

If you merely want to know whether the `HttpSession` exists, you can instead inject the `HttpSessionStatus` bean that Seam Servlet provides.

3.7. @Inject HttpSessionStatus

The `HttpSessionStatus` is a request-scoped bean that provides access to the status of the `HttpSession`. It can be injected safetly into any CDI bean as follows:

```
@Inject
private HttpSessionStatus sessionStatus;
```

You can invoke the `isActive()` method to check if the session has been created, and the `getSession()` method to retrieve the `HttpSession`, which will be created if necessary.

```
if (!sessionStatus.isActive()) {
    System.out.println("Session does not exist. Creating it now.");
    HttpSession session = sessionStatus.get();
    assert session.isNew();
}
```

3.8. @Inject @ContextPath

The context path is made available as a dependent-scoped bean. It can be injected safetly into any request-scoped CDI bean as follows:

```
@Inject @ContextPath
private String contextPath;
```

You can safetly inject the context path into a bean with a wider scope using an instance provider:

```
@Inject @ContextPath
private Instance<String> contextPathProvider;
...
String contextPath = contextPathProvider.get();
```

The context path is retrieved from the `HttpServletRequest`.

3.9. @Inject List<Cookie>

The list of `Cookie` objects is made available as a request-scoped bean. It can be injected safetly into any CDI bean as follows:

```
@Inject
private List<Cookie> cookies;
```

The producer uses a reference to the request-scoped `HttpServletRequest` bean to retrieve the `Cookie` intances by calling `getCookie()`.

3.10. @Inject @CookieParam

Similar to the `@RequestParam`, you can use the `@CookieParam` qualifier to inject an HTTP header parameter. Here's an example of how you inject the username of the last logged in user (assuming you have previously stored it in a cookie):

```
@Inject @CookieParam
private String username;
```

If the type at the injection point is `Cookie`, the `Cookie` object will be injected instead of the value.

```
@Inject @CookieParam
private Cookie username;
```

The `@CookieParam` also support a default value using the `@DefaultValue` annotation.

Warning

Since the bean produced is dependent-scoped, use of the `@CookieParam` annotation on class fields and bean properties is only safe for request-scoped beans. Beans with wider scopes should wrap this bean in an `Instance` bean and retrieve the value within context of the thread in which it's needed.

```
@Inject @CookieParam("username")
private Instance<String> usernameResolver;
...
String username = usernameResolver.get();
```

3.11. @Inject @ServerInfo

The server info string is made available as a dependent-scoped bean. It can be injected safetly into any CDI bean as follows:

```
@Inject @ServerInfo
private String serverInfo;
```

The context path is retrieved from the `ServletContext`.

3.12. @Inject @Principal

The security `Principal` for the current user is made available by CDI as an injectable resource (not provided by Seam Servlet). It can be injected safetly into any CDI bean as follows:

```
@Inject
private Principal principal;
```

Exception handling: Seam Catch integration

Seam Catch provides a simple, yet robust foundation for modules and/or applications to establish a customized exception handling process. Seam Servlet ties into the exception handling model by forwarding all unhandled Servlet exceptions to Catch so that they can be handled in a centralized, extensible and uniform manner.

4.1. Background

The Servlet API is extremely weak when it comes to handling exceptions. You are limited to handling exceptions using the built-in, declarative controls provided in web.xml. Those controls give you two options:

- send an HTTP status code
- forward to an error page (servlet path)

To make matters more painful, you are required to configure these exception mappings in web.xml. It's really a dinosaur left over from the past. In general, the Servlet specification seems to be pretty non-chalant about exceptions, telling you to "handle them appropriately." But how?

That's where the Catch integration in Seam Servlet comes in. The Catch integration traps all unhandled exceptions (those that bubble outside of the Servlet and any filters) and forwards them on to Catch. Exception handlers are free to handle the exception anyway they like, either programmatically or via a declarative mechanism.

If a exception handler registered with Catch handles the exception, then the integration closes the response without raising any additional exceptions. If the exception is still unhandled after Catch finishes processing it, then the integration allows it to pass through to the normal Servlet exception handler.

4.2. Defining a exception handler for a web request

You can define an exception handler for a web request using the normal syntax of a Catch exception handler. Let's catch any exception that bubbles to the top and respond with a 500 error.

```
@HandlesExceptions
public class ExceptionHandlers {
   void handleAll(@Handles CaughtException<Throwable> caught, HttpServletResponse response) {
      response.sendError(500, "You've been caught by Catch!");
   }
```

```
}
```

That's all there is to it! If you only want this handler to be used for exceptions raised by a web request (excluding web service requests like JAX-RS), then you can add the `@WebRequest` qualifier to the handler:

```
@HandlesExceptions
public class ExceptionHandlers {
   void handleAll(@Handles @WebRequest
         CaughtException<Throwable> caught, HttpServletResponse response) {
      response.sendError(500, "You've been caught by Catch!");
   }
}
```

Note

Currently, `@WebRequest` is required to catch exceptions initiated by the Servlet integration because of a bug in Catch.

Let's consider another example. When the custom `AccountNotFound` exception is thrown, we'll send a 404 response using this handler.

```
void handleAccountNotFound(@Handles @WebRequest
      CaughtException<AccountNotFound> caught, HttpServletResponse response) {
   response.sendError(404, "Account not found: " + caught.getException().getAccountId());
}
```

In a future release, Seam Servlet will include annotations that can be used to configure these responses declaratively.

Retrieving the BeanManager from the servlet context

Typically, the `BeanManager` is obtained using some form of injection. However, there are scenarios where the code being executed is outside of a managed bean environment and you need a way in. In these cases, it's necessary to lookup the `BeanManager` from a well-known location.

Warning

In general, you should isolate external `BeanManager` lookups to integration code.

The standard mechanism for locating the `BeanManager` from outside a managed bean environment, as defined by the JSR-299 specification, is to look it up in JNDI. However, JNDI isn't the most convenient technology to depend on when you consider all popular deployment environments (think Tomcat and Jetty).

As a simpler alternative, Seam Servlet binds the `BeanManager` to the following servlet context attribute (whose name is equivalent to the fully-qualified class name of the `BeanManager` interface:

```
javax.enterprise.inject.spi.BeanManager
```

Seam Servlet also includes a provider that retrieves the `BeanManager` from this location. Anytime the Seam Servlet module needs a reference to the `BeanManager`, it uses this lookup mechanism to ensure that the module works consistently across deployment environments, especially in Servlet containers.

You can retrieve the `BeanManager` in the same way. If you want to hide the lookup, you can extend the `BeanManagerAware` class and retrieve the `BeanManager` from the the method `getBeanManager()`, as shown here:

```
public class NonManagedClass extends BeanManagerAware {
   public void fireEvent() {
      getBeanManager().fireEvent("Send me to a managed bean");
   }
}
```

Alternatively, you can retrieve the `BeanManager` from the method `getBeanManager()` on the `BeanManagerLocator` class, as shown here:

```
public class NonManagedClass {
   public void fireEvent() {
      new BeanManagerLocator().getBeanManager().fireEvent("Send me to a managed bean");
   }
}
```

Tip

The best way to transfer execution of the current context to the managed bean environment is to send an event to an observer bean, as this example above suggests.

Under the covers, these classes look for the `BeanManager` in the servlet context attribute covered in this section, amongst other available strategies. Refer to the *BeanManager provider* chapter of the Seam Solder reference guide for information on how to leverage the servlet context attribute provider to access the `BeanManager` from outside the CDI environment.