Seam Catch

1. Seam Catch Introduction 1
2. Installation 3
3. Usage 5
3.1. Annotations 5
3.1.1. @HandlesExceptions 5
3.1.2. @Handles 6
3.2. Adding Handlers 6
3.3. Ordering of Handlers 6
3.4. Traversal of the causing container 6
3.5. API Objects 7
3.5.1. CaughtException 7
3.5.2. ExceptionStack 7
4. Framework Integration 9
4.1. Creating and Firing an ExceptionToCatchEvent 9
4.2. Default Handlers and Qualifiers 9
4.2.1. Default Handlers 9
4.2.2. Qualifiers 9
4.3. Supporting ServiceHandlers 10

Seam Catch Introduction

The Seam Catch module creates a simple, yet robust base for other modules and users to create a custom and complete exception handling process. Exception handling is done using CDI events, keeping exception handling noninvasive and also helping the program or module to stay minimally coupled to the exception handling framework.

Installation

The Seam Catch API is the only compile time dependency a project needs, and an implementation must also be included, either explicitly or via some other module depending on it (and exposing their own specialized extensions) is all that is needed during runtime. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your pom.xml file:

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

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

Note

The runtime dependency is only needed if another Seam 3 module being used doesn't already use it. Typically this will only be for Java SE development.

Tip

Replace ${seam-catch-version} with the most recent or appropriate version of Seam Catch.

Usage

An end user of the Seam Catch Framework is typically only concerned with Exception Handlers (methods in Handler Beans, which are similar to CDI Observers). Handler Beans are CDI beans with the `@HandlesExceptions` annotation. There may be other resources made available by other modules which can be injected into handler methods on an as needed basis. For further information, please check the API docs, or examples.

3.1. Annotations

3.1.1. @HandlesExceptions

The `@HandlesException` annotation is simply a marker annotation instructing the Seam Catch CDI extension to scan the bean for handler methods.

Example

```
@HandlesExceptions
public class MyHandlers
{
   public void catchAllHandler(@Handles(during = TraversalPath.DESCENDING) @MyFramework CaughtExceptio
   {
      log.warn("Exception occured: " + event.getException().getMessage());
   }
}
```

This is a complete and valid handler showing all the current features of handlers. The Handler Bean is defined by the class level annotation `@HandlesExceptions` and the actual handler is defined by the method that takes a `CaughtException` of type `Throwable` which is annotated using the `@Handles` annotation. Also notice the handler is qualified using the `@MyFramework` qualifier. This works the same as qualifers in CDI Observers, it will only be invoked for exceptions (it catches type`Throwable`) where the initial `ExceptionToCatchEvent` was created with the `@MyFramework` annotation passed to the constructor. The Logger instance is also injected into the handler when it is invoked. The handler has a default precedence of 0 and a `TraversalPath` of `DESCENDING`. It does not modify flow control of other handlers however and simply uses the default proceed.

Note

This annotation may be deprecated favoring annotation indexing done by Seam Solder.

3.1.2. @Handles

`@Handles` is a parameter annotation that specifies a method as an exception handler. It acts similar to the `@Observes` annotation from CDI. In addition to promoting a normal method to an exception handler it also carries data about the handler:

- a precedence relative to other handlers of the same type (the heigher the precendence, the closer to the top of the causing traversal it is placed, relative to other handlers for the same exception type, zero being the default)
- the direction of the traversal path during which the handler is invoked (`TraversalPath.ASCENDING` being default)

The `@Handles` annotation must be placed on the first parameter of the method, which must be of type CaughtException. Handler methods are similar to CDI Obeservers and follow the same principals and guidelines (such as invocation, injection of parameters, qualifiers, etc). They differ from Observer methods in that:

- they are ordered before they are invoked
- the first parameter of the method must be a generized CaughtException object

A handler is guaranteed to only be invoked once per exception (unless it is unmuted via the CaughtException object by calling`unMute()`). Handlers must not throw checked exceptions, and should avoid throwing unchecked exceptions.

3.2. Adding Handlers

Adding a handler is simply creating a class and a method the follows the above rules (class annotated with `@HandlesExceptions` and a method with the first parameter being a CaughtException and annotated with`@Handles`). Catch will discover all handler methods at deploy time. See the example above for a simple, but complete handler.

3.3. Ordering of Handlers

The ordering of handlers is multifaceted. Based on the traversal path of the causing container handlers are ordered according to the hiearchy of the excption type (most specific first if `TraversalPath.ASCENDING`, least specific first if `TraversalPath.DESCENDING` traversal), and the precedence when two handlers are for the same exception type.

The `precedence` of a handler helps determine the order of the handler relative to other handlers of the same exception type. It follows a high-to-low integer schema (the higher the precedence, the sooner the handler is invoked during traversal of the causing chain).

3.4. Traversal of the causing container

When an exception is handled with Catch the causing container is unwrapped to get at each exception. The first pass (`TraversalPath.DESCENDING`) starts with the outer most exception

working it's way to the root exception. The traversal is then reversed and traversed from root cause up. This allows handlers to take part in various stages of the causing container. At each entry in the container, handlers are invoked based on the exception type (either an exact match or a super type) of the entry. For example if the exception type is`SocketException`, handlers for types `SocketException`, `IOException`, `Exception` and `Throwable` would all invoked (in that order), however, a handler for `BindException` would not be invoked.

3.5. API Objects

There are other objects used in Catch that should be familiar to handler writers namely

- `CaughtException`
- `ExceptionStack`

3.5.1. CaughtException

`CaughtException` contains methods to interact with the handling procces, allowing a level of flow control to be available to handler (such as re-throwing the exception, or aborting), and allowing a handler to be unmuted. Once a handler is invoked it is muted, meaning it will not be run again for that causing container, unless it is explicitly marked as unmuted via the `CaughtException.unMute()` object.

Five methods exist on the `CaughtException` object to give flow control to the handler

- `abort()` - terminate all handling immediately after this handler, does not mark the exception as handled, does not re-throw the exception.
- `rethrow()` - continues through all handlers, but once all handlers have been called (assuming another handler does not call abort() or handled()) the initial exception passed to Catch is rethrown. Does not mark the exception as handled.
- `handled()` - marks the exception as handled and terminates further handling.
- `proceed()` - default. Marks the exception as handled and proceeds with the rest of the handlers.
- `proceedToCause()` - marks the exception as handled, but proceeds to the next cause in the cause container, without calling other handlers for the current cause.

3.5.2. ExceptionStack

`ExceptionStack` contains information about the exception causes relative to the current cause. Please see API docs for more information, all methods are fairly self-explanatory.

Framework Integration

Integration of Seam Catch with other frameworks consists of one main step, and two other optional (but highly encouraged) step:

- creating and firing an `ExceptionToCatchEvent`
- adding any default handlers and qualifiers with annotation literals (optional)
- supporting ServiceHandlers for creating exception handlers

4.1. Creating and Firing an ExceptionToCatchEvent

An `ExceptionToCatchEvent` is constructed by passing a `Throwable` and optionally qualifiers for handlers. Firing the event is done via CDI events (either straight from the `BeanManager` or injecting a `Event<ExceptionToCatchEvent>` and calling fire).

To ease the burden on the application developers, the integration should tie into the exception handling mechanism of the integrating framework, if any exist. By tying into the framework's exception handling, any uncaught exceptions should be routed through the Seam Catch system and allow handlers to be invoked. This is the typical way of using the Seam Catch framework. Of course, it doesn't stop the application developer from firing their own `ExceptionToCatchEvent` within a catch block.

4.2. Default Handlers and Qualifiers

4.2.1. Default Handlers

An integration with Catch can define it's own handlers to always be used. It's recommended that any built-in handler from an integration have a very low precedence, be a handler for as generic an exception as is suitable (i.e. Seam Persistence could have a built-in handler for PersistenceExceptions to rollback a transaction, etc), and make use of qualifiers specific for the integration. This helps limit any collisions with handlers the application developer may create.

> **Note**
> Hopefully at some point there will be a way to conditionally enable handlers so the application developer will be able to selectively enable any default handlers. Currently this does not exist, but is something that will be explored.

4.2.2. Qualifiers

Catch supports qualifiers for the`CaughtException`. To add a qualifier to be used when firing handlers they must be add to the `ExceptionToCatchEvent` via the constructor (please see API

docs for more info). Qualifiers for integrations should be used to avoid collisions in the application error handling both when defining handlers and when firing events from the integration.

4.3. Supporting ServiceHandlers

ServiceHandlers [http://docs.jboss.org/weld/extensions/reference/latest/en-US/html_single/#servicehandler] make for a very easy and concise way to define exception handlers take the following example comes from the jaxrs example in the distribution:

```
@HandlesExceptions
@ExceptionResponseService
public interface DeclarativeRestExceptionHandlers
{

    @SendHttpResponse(status = 403, message = "Access to resource denied (Annotation-configured response)")
  void onNoAccess(@Handles @RestRequest CaughtException<AccessControlException> e);

    @SendHttpResponse(status = 400, message = "Invalid identifier (Annotation-configured response)")
  void onInvalidIdentifier(@Handles @RestRequest CaughtException<IllegalArgumentException> e);
}
```

All the vital information that would normally be done in the handler method is actually contained in the `@SendHttpResponse` annotation. The only thing left is some boiler plate code to setup the `Response`. In a jax-rs application (or even in any web application) this approach helps developers cut down on the amount of boiler plate code they have to write in their own handlers and should be implemented in any Catch integration, however, there may be situtations where ServiceHandlers simply do not make sense.

Note

If ServiceHandlers are implemented make sure to document if any of the methods are called from `CaughtException`, specifically `abort()`, `handled()` or `rethrow()`. These methods affect invocation of other handlers (or rethrowing the exception in the case of `rethrow()`).