Seam Catch

1. Introduction 1
2. Installation 3
2.1. Maven dependency configuration 3
3. Usage 5
3.1. Exception handlers 5
3.2. Exception handler annotations 5
3.2.1. @HandlesExceptions 5
3.2.2. @Handles 6
3.3. Exception stack trace processing 8
3.4. Exception handler ordering 8
3.4.1. Traversal of exception type hierarchy 9
3.4.2. Handler precendence 10
3.5. APIs for exception information and flow control 11
3.5.1. CaughtException 11
3.5.2. ExceptionStack 12
4. Framework Integration 13
4.1. Creating and Firing an ExceptionToCatch event 13
4.2. Default Handlers and Qualifiers 13
4.2.1. Default Handlers 13
4.2.2. Qualifiers 13
4.3. Supporting ServiceHandlers 14

Introduction

Exceptions are a fact of life. As developers, we need to be prepared to deal with them in the most graceful manner possible. Seam Catch provides a simple, yet robust foundation for modules and/or applications to establish a customized exception handling process. By employing a delegation model, Catch allow exceptions to be addressed in a centralized, extensible and uniform manner.

Catch is first notified of an exception to be handled via a CDI event. This event is fired either by the application or a Catch integration. Catch then hands the exception off to a chain of registered handlers, which deal with the exception appropriately. The use of CDI events to connect exceptions to handlers makes this strategy of exception handling non-invasive and minimally coupled to Catch's infrastructure.

The exception handling process remains mostly transparent to the developer. In some cases, you register an exception handler simply by annotating a handler method. Alternatively, you can handle an exception programmatically, just as you would observe an event in CDI.

In this guide, we'll explore the various options you have for handling exceptions using Catch, as well as how framework authors can offer Catch integration.

Installation

To use the Seam Catch module, you need to add the Seam Catch API to your project as a compile-time dependency. At runtime, you'll also need the Seam Catch implementation, which you either specify explicitly or through a transitive dependency of another module that depends on it (as part of exposing its own Catch integration).

First, check your application's library dependencies to see whether Seam Catch is already being included by another module (such as Seam Servlet). If not, you'll need to setup the dependencies as described below.

2.1. 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 Catch:

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

Tip

Substitute the expression ${seam.catch.version} with the most recent or appropriate version of Seam Catch. 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.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>
```

Now you're ready to start catching exceptions!

Usage

3.1. Exception handlers

As an application developer (i.e., an end user of Catch), you'll be focused on writing exception handlers. An exception handler is a method on a CDI bean that is invoked to handle a specific type of exception. Within that method, you can implement any logic necessary to handle or respond to the exception.

Given that exception handler beans are CDI beans, they can make use of dependency injection, be scoped, have interceptors or decorators and any other functionality available to CDI beans.

Exception handler methods are designed to follow the syntax and semantics of CDI observers, with some special purpose exceptions explained in this guide. The advantage of this design is that exception handlers will be immediately familiar to you if you are studying or well-versed in CDI.

In this chapter, you'll learn how to define an exception handler and explore how and when it gets invoked. We'll begin by covering the two annotations that are used to declare an exception handler, `@HandlesExceptions` and `@Handles`.

3.2. Exception handler annotations

Exception handlers are contained within exception handler beans, which are CDI beans annotated with `@HandlesExceptions`. Exception handlers are methods whose first parameter is an instance of `CaughtException<T extends Throwable>` annotated with the `@Handles` annotation.

3.2.1. @HandlesExceptions

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

Let's designate a CDI bean as an exception handler by annotating it with `@HandlesException`.

```
@HandlesExceptions
public class MyHandlers {}
```

That's all there is to it. Now we can begin defining exception handling methods on this bean.

Note

The `@HandlesExceptions` annotation may be deprecated in favor of annotation indexing done by *Seam Solder*.

3.2.2. @Handles

`@Handles` is a method parameter annotation that designates a method as an exception handler. Exception handler methods are registered on beans annotated with `@HandlesExceptions`. Catch will discover all such methods at deployment time.

Let's look at an example. The following method is invoked for every exception that Catch processes and prints the exception message to stout. (`Throwable` is the base exception type in Java and thus represents all exceptions).

```
@HandlesExceptions                                          (1)
public class MyHandlers
{
  void printExceptions(@Handles CaughtException<Throwable> evt)          (2)
  {
    System.out.println("Something bad happened: " +
        evt.getException().getMessage());                   (3)
    evt.proceed();                                          (4)
  }
}
```

1. The `@HandlesExceptions` annotation signals that this bean contains exception handler methods.
2. The `@Handles` annotation on the first parameter designates this method as an exception handler. This parameter must be of type `CaughtException<T extends Throwable>`, otherwise it's detected as a definition error. The type parameter designates which exception the method should handle. This method is notified of all exceptions (requested by the base exception type `Throwable`).
3. The `CaughtException` instance provides access to information about the exception and can be used to control exception handling flow. In this case, it's used to read the current exception being handled in the exception stack trace, as returned by `getException()`.
4. This handler does not modify the invocation of subsequent handlers, as designated by invoking `proceed()` on `CaughtException`. As this is the default behavior, this line could be omitted.

The `@Handles` annotation must be placed on the first parameter of the method, which must be of type `CaughtException<T extends Throwable>`. Handler methods are similar to CDI observers and, as such, follow the same principals and guidelines as observers (such as invocation, injection of parameters, qualifiers, etc) with the following exceptions:

- the first parameter of a handler method must be a `CaughtException`

- handlers are ordered before they are invoked (invocation order of observers is non-deterministic)
- any handler can prevent subsequent handlers from being invoked

In addition to designating a method as exception handler, the `@Handles` annotation specifies two pieces of information about when the method should be invoked relative to other handler methods:

- a precedence relative to other handlers for the same exception type. Handlers with higher precendence are invoked before handlers with lower precendence that handle the same exception type. The default precendence (if not specified) is 0.
- the direction of the traversal path (i.e., phase) during which the handler is invoked. The default traversal direction (if not specified) is `TraversalPath.ASCENDING`.

Let's take a look at more sophisticated example that uses all the features of handlers to log all exceptions.

```
@HandlesExceptions                                                    (1)
public class MyHandlers
{
  void logExceptions(@Handles(during = TraversalPath.DESCENDING)      (2)
       @WebRequest CaughtException<Throwable> evt,                    (3)
       Logger log)                                                    (4)
  {
    log.warn("Something bad happened: " + evt.getException().getMessage());
  }
}
```

(1) The `@HandlesExceptions` annotation signals that this bean contains exception handler methods.

(2) This handler has a default precedence of 0 (the default value of the precedence attribute on `@Handles`). It's invoked during the descending traversal path. For more information on traversal, see the section *Section 3.4.1, "Traversal of exception type hierarchy"*.

(3) This handler is qualified with `@WebRequest`. When Catch calculates the handler chain, it filters handlers based on the exception type and qualifiers. This handler will only be invoked for exceptions passed to Catch that carry the `@WebRequest` qualifier. We'll assume this qualifier distinguishes a web page request from a REST request.

(4) Any additional parameters of a handler method are treated as injection points. These parameters are injected into the handler when it is invoked by Catch. In this case, we are injecting a `Logger` bean that must be defined within the application (or by an extension).

A handler is guaranteed to only be invoked once per exception (automatically muted), unless it reenables itself by invoking the `unMute()` method on the `CaughtException` instance.

Handlers must not throw checked exceptions, and should avoid throwing unchecked exceptions.

3.3. Exception stack trace processing

When an exception is thrown, chances are it's nested (wrapped) inside other exceptions. (If you've ever examined a server log, you'll appreciate this fact). The collection of exceptions in its entirety is termed an exception stack trace.

The outermost exception of an exception stack trace (e.g., EJBException, ServletException, etc) is probably of little use to exception handlers. That's why Catch doesn't simply pass the exception stack trace directly to the exception handlers. Instead, it intelligently unwraps the stack trace and treats the root exception cause as the primary exception.

The first exception handlers to be invoked by Catch are those that match the type of root cause. Thus, instead of seeing a vague `EJBException`, your handlers will instead see an meaningful exception such as `ConstraintViolationException`. *This feature, alone, makes Catch a worthwhile tool.*

Catch continues to work through the exception stack trace, notifying handlers of each exception in the stack, until a handler flags the exception as handled. Once an exception is marked as handled, Catch stops processing the exception. If a handler instructed Catch to rethrow the exception (by invoking `CaughtException#rethrow()`, Catch will rethrow the exception outside the Catch infrastructure. Otherwise, it simply returns flow control to the caller.

Consider a stack trace containing the following nested causes (from outer cause to root cause):

- EJBException
- PersistenceException
- SQLGrammarException

Catch will unwrap this exception and notify handlers in the following order:

1. SQLGrammarException
2. PersistenceException
3. EJBException

If there's a handler for `PersistenceException`, it will likely prevent the handlers for `EJBException` from being invoked, which is a good thing since what useful information can really be obtained from `EJBException`?

3.4. Exception handler ordering

While processing one of the causes in the exception stack trace, Catch has a specific order it uses to invoke the handlers, operating on two axes:

- traversal of exception type hierarchy
- relative handler precedence

We'll first address the traversal of the exception type hierarchy, then cover relative handler precedence.

3.4.1. Traversal of exception type hierarchy

Catch doesn't simply invoke handlers that match the exact type of the exception. Instead, it walks up and down the type hierarchy of the exception. It first notifies least specific handler, then gradually works down the type hiearchy towards handlers for the actual exception type. It then walks back up again towards the least specific handler.

There are two phases of this traversal:

- ascending
- descending

By default, handlers are registered into the ascending traversal path. That means in most cases, Catch starts with handlers of the actual exception type and works up towards the handler for the least specific type.

However, when a handler is registered to be notified during the descending traversal, as in the example above, Catch will notify that exception handler before the exception handler for the actual type is notified.

Let's consider an example. Assume that Catch is handling the `SocketException`. It will notify handlers in the following order:

1. `Throwable`
2. `Exception`
3. `IOException`
4. `SocketException`
5. `IOException`
6. `Exception`
7. `Throwable`

The same type traversal occurs for each exception processed in the stack trace.

In order for a handler to be notified of the `IOException` before the `SocketException`, it would have to specify the descending traversal path explicitly:

```
void handleIOException(@Handles(during = TraversalPath.DESCENDING)
    CaughtException<IOException> evt)
{
  System.out.println("An I/O exception occurred, but not sure what type yet");
}
```

Descending handlers are typically used for logging exceptions because they are not likely to be short-circuited (and thus always get invoked).

3.4.2. Handler precendence

When Catch finds more than one handler for the same exception type, it orders the handlers by precendence. Handlers with higher precendence are executed before handlers with a lower precedence. If Catch detects two handlers for the same type with the same precedence, it detects it as an error and throws an exception at deployment time.

Let's define two handlers with different precendence:

```
void handleIOExceptionFirst(@Handles(precendence = 100) CaughtException<IOException> evt)
{
  System.out.println("Invoked first");
}

void handleIOExceptionSecond(@Handles CaughtException<IOException> evt)
{
  System.out.println("Invoked second");
}
```

The first method is invoked first since it has a higher precendence (100) than the second method, which has the default precedence (0).

To make specifying precendence values more convenience, Catch provides several built-in constants, available on the `Precedence` interface:

- BUILT_IN = -100
- FRAMEWORK = -50
- DEFAULT = 0

- LOW = 50
- HIGH = 100

To summarize, here's how Catch determines the order of handlers to invoke (until a handler marks exception as handled):

1. Unwrap exception stack
2. Begin processing root cause
3. Find handler for least specific handler marked for descending traversal
4. If multiple handlers for same type, invoke handlers with higher precendence first
5. Continue above steps for each exception in stack
6. Return to root cause
7. Find handler for most specific handler marked for ascending traversal
8. If multiple handlers for same type, invoke handlers with higher precendence first
9. Continue previous two steps for each exception in stack

3.5. APIs for exception information and flow control

There are two APIs provided by Catch that should be familiar to application developers:

- `CaughtException`
- `ExceptionStack`

3.5.1. CaughtException

In addition to providing information about the exception being handled, the `CaughtException` object contains methods to control the exception handling process, such as rethrowing the exception, aborting the handler chain or unmuting the current handler.

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.

Once a handler is invoked it is muted, meaning it will not be run again for that exception stack trace, unless it's explicitly marked as unmuted via the `unmute()` method on `CaughtException`.

3.5.2. ExceptionStack

`ExceptionStack` contains information about the exception causes relative to the current exception cause. It is accessed by calling the method `getCauses()` on the `CaughtException` object. 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) steps:

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

4.1. Creating and Firing an ExceptionToCatch event

An `ExceptionToCatch` 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<ExceptionToCatch>` 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 `ExceptionToCatch` 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 `ExceptionToCatch` 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/seam/3/solder/latest/reference/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()`).