Seam 3

Bundled Reference Guide

Credits

1.1. List of contributors

Seam is a collaborative project created by the Open Source community. We would like to thank all of the following people for their contributions, without which Seam would not have been possible.

- Gavin King (Project Founder)

- Shane Bryzak (Project Lead)

- Pete Muir (former Project Lead)

- Dan Allen (Community Liaison)

- John Ament

- Max Rydahl Andersen

- Jay Balunas

- Christian Bauer

- Lincoln Baxter III

- Emmanuel Bernard

- Mike Brock

- Stuart Douglas

- Matt Drees

- Ken Finnigan

- Denis Forveille

- Jose Freitas

- Jordan Ganoff

- George Gastaldi

- Martin Gencur

- Jozef Hartinger

- Peter Hilton

- Ales Justin

- Nicklas Karlsson

- Marcel Kolsteren

- Brian Leathem

- Cody Lerum

- Gunnar Morling

- Marek Novotny

- Jacob Orshalick

- James Perkins

- Clint Popetz

- Jason Porter

- Karel Piwko

- Ove Ranheim

- Norman Richards

- Daniel Roth

- Antoine Sabot-Durand

- Sebastian Sachtleben

- Amir Sadrinia

- Marek Schmidt

- Ondrej Skutka

- Marek Smigielski

- Mark Struberg

- Tihomir Surdilovic

- Mike Youngstrom

- Michael Yuan

1.2. Would you like to contribute?

We are looking for talented people to help us in making Seam the best application framework in the world. Seam is an Open Source project with an extensive developer and user community, consisting of both full time and volunteer team members from all over the world.

There are many ways to contribute, such as:

- Providing code, such as bug fixes and enhancements, or entirely new features

- Improving this documentation - even minor things like typos and grammar correction fixes are appreciated.

- Participate on the user forums to share your Seam knowledge with other users

- Writing a blog or article about a particular feature of Seam

If you would like to be involved in the ongoing development of Seam please visit us at *http://www.seamframework.org/Seam3/Contribute* to find out more.

Seam

2.1. Overview

Seam's mission is to provide a fully-integrated development platform for building rich, standards-based Internet applications tailored for traditional and cloud deployments.

The Seam 3 project is organized as a collection of modules and developer tooling tailored for Java EE 6 application development, built on top of the component model defined by *JSR-299 Context and Dependency Injection* (CDI). CDI is a JCP standard, you can find out more about it at *http://jcp.org/en/jsr/summary?id=299*.

Seam's modules leverage portable CDI extensions to build on the core Java EE functionality and integrate with JBoss and third-party projects. Together, these modules provide many of the popular features and integrations from Seam 2 (security, internationalization, JSF, rules, BPM) while also exploring new integrations and designs.

The developer tooling for Seam is provided by JBoss Tools and Seam Forge. JBoss Tools enhances Eclipse with features designed to help developers write, test and deploy enterprise Java applications. Seam Forge is an incremental project enhancement API and shell.

This guide steps you through the modules and select tooling, covering the purpose, APIs and usage scenarios for each. Collectively, this software should give you everything you need to develop comprehensive, robust and compelling enterprise applications.

2.2. Seam Bill of Materials

The Seam 3 build is based on Maven 3. Each Seam module is a separate project, with its own release cycle. Each Seam module is a multi-module project contains the api, implementation, examples and documentation. Select modules are assembled together to create a Seam distribution, or stack release.

To keep the modules in sync, the Seam project publishes a special Maven POM known as a "Bill of Materials" (BOM), which we'll refer to as the Seam BOM. The Seam BOM defines the versions of all the Seam modules and third-party libraries that are used in the Seam stack using Maven's dependency management facility.

You can import these version definitions into your project by adding the Seam BOM as a dependency with scope `import`. The benefit of doing so is that it relieves you from having to specify the version of any Seam module explicitly. It also means you can upgrade all your Seam modules at once by just updating the version of the BOM.

Generally, the easiest way to accomplish this import is by first defining a property for the Seam BOM version:

```
<properties>
```

```
    <seam.version>3.0.0.Final</seam.version>
</properties>
```

Then you add the following dependency declaration to the `dependencyManagement` section of your project's POM file (or parent POM, if you use one).

```
<dependencyManagement>
    <dependencies>
      <dependency>
          <groupId>org.jboss.seam</groupId>
          <artifactId>seam-bom</artifactId>
          <version>${seam.version}</version>
          <type>pom</type>
          <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
```

Then, it's a simple matter to declare which Seam module dependencies your project requires by adding them inside the `dependencies` section. There's no need to specify a version of the module as it gets inherited from the Seam BOM.

```
<dependency>
   <groupId>org.jboss.seam.solder</groupId>
   <artifactId>seam-solder</artifactId>
</dependency>
```

To see which version is going to get selected, use the dependency analysis tools in Maven:

```
mvn dependency:tree
```

You may upgrade an individual module by specifying the version explicitly. There's no crime in doing so. The Seam BOM is there as a convenience and a reference point for the recommended module version matrix. It's up to you how closely to follow it.

Each of the Seam modules also use the Seam BOM to keep the versions of related modules in sync. Once in a while, a module may specify a version of another module that's different from the Seam BOM. We usually try to get this worked out by the time we make a Seam stack release to fix the version matrix.

Refer to the *Build System Architecture* [http://seamframework.org/Seam3/ BuildSystemArchitecture] page on the Seam website for more detail about how the Seam 3 project is structured. Though, for the purpose of using Seam, how to import the module artifacts is likely all you need to know about the project's build.

Part I. Solder

Introduction

Solder is a library of Generally Useful Stuff (TM), particularly if you are developing an application based on CDI (JSR-299 Java Contexts and Dependency Injection), or a CDI based library or framework.

This guide is split into three parts. *???* details extensions and utilities which are likely to be of use to any developer using CDI; *???* describes utilities which are likely to be of use to developers writing libraries and frameworks that work with CDI; *???* discusses extensions which can be used to implement configuration for a framework

Getting Started

Getting started with Solder is easy. All you need to do is put the API and implementation JARs on the classpath of your CDI application. The features provided by Solder will be enabled automatically.

Some additional configuration, covered at the end of this chapter, is required if you are using a pre-Servlet 3.0 environment.

3.1. Maven dependency configuration

If you are using *Maven* [http://maven.apache.org/] as your build tool, first make sure you have configured your build to use the *JBoss Community repository* [http://community.jboss.org/wiki/MavenGettingStarted-Users], where you can find all the Seam artifacts. Then, add the following dependencies to your `pom.xml` file to get started using Solder:

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

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

> **Tip**
>
> Substitute the expression ${solder.version} with the most recent or appropriate version of Solder. Alternatively, you can create a *Maven user-defined property* [http://www.sonatype.com/books/mvnref-book/reference/resource-filtering-sect-properties.html#resource-filtering-sect-user-defined] to satisfy this substitution so you can centrally manage the version.

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

3.2. Transitive dependencies

Most of Solder has very few dependencies, only one of which is not provided by Java EE 6:

- `javax.enterprise:cdi-api` (provided by Java EE 6)

- `javax.inject:javax:inject` (provided by Java EE 6)

- `javax.annotation:jsr250-api` (provided by Java EE 6)

- `javax.interceptor:interceptor-api` (provided by Java EE 6)

- `javax.el:el-api` (provided by Java EE 6)

> **Tip**
>
> The POM for Solder specifies the versions required. If you are using Maven 3, you can easily import the `dependencyManagement` into your POM by declaring the following in your `depdendencyManagement` section:
>
> ```xml
> <dependency>
>   <groupId>org.jboss.solder</groupId>
>   <artifactId>seam-solder-impl</artifactId>
>   <version>${solder.version}</version>
>   <type>pom</type>
>   <scope>import</scope>
> </dependency>
> ```

Some features of Solder require additional dependencies (which are declared optional, so will not be added as transitive dependencies):

`org.javassist:javassist`
: Service Handlers, Unwrapping Producer Methods

`javax.servlet:servlet-api`
: Accessing resources from the Servlet Context

In addition, a logger implementation (SLF4J, Log4J, JBoss Log Manager or the JDK core logging facility) is required. Refer to *Chapter 8, Logging, redesigned* for more information about how logging is handled in Solder.

3.3. 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 a Servlet component in your application's web.xml to access resources from the Servlet Context.

```
<listener>
  <listener-class>org.jboss.solder.resourceLoader.servlet.ResourceListener</listener-class>
</listener>
```

This registration happens automatically in a Servlet 3.0 environment through the use of a /META-INF/web-fragment.xml included in the Solder implementation.

You're all setup. It's time to dive into all the useful stuff that Solder provides!

Enhancements to the CDI Programming Model

Solder provides a number enhancements to the CDI programming model which are under trial and may be included in later releases of *Contexts and Dependency Injection*.

4.1. Preventing a class from being processed

4.1.1. @Veto

Annotating a class `@Veto` will cause the type to be ignored, such that any definitions on the type will not be processed, including:

- the managed bean, decorator, interceptor or session bean defined by the type

- any producer methods or producer fields defined on the type

- any observer methods defined on the type

For example:

```
@Veto
class Utilities {
    ...
}
```

Besides, a package can be annotated with `@Veto`, causing all beans in the package to be prevented from registration.

Example 4.1. package-info.java

```
@Veto
package com.example;

import org.jboss.solder.core.Veto;
```

> **Note**
>
> The `ProcessAnnotatedType` container lifecycle event will be called for vetoed types.

4.1.2. @Requires

Annotating a class with @Requires will cause the type to be ignored if the class dependencies cannot be satisfied. Any definitions on the type will not be processed:

- the managed bean, decorator, interceptor or session bean defined by the type

- any producer methods or producer fields defined on the type

- any observer methods defined on the type

> **Tip**
>
> Solder will use the Thread Context ClassLoader, as well as the classloader of the type annotated @Requires to attempt to satisfy the class dependency.

For example:

```
@Requires("javax.persistence.EntityManager")
class EntityManagerProducer {

  @Produces
  EntityManager getEntityManager() {
    ...
  }

}
```

Annotating a package with @Requires causes all beans in the package to be ignored if the class dependencies cannot be satisfied. If both a class and it's package are annotated with @Requires, both package-level and class-level dependencies have to be satisfied for the bean to be installed.

> **Note**
>
> The ProcessAnnotatedType container lifecycle event will be called for required types.

4.2. @Exact

Annotating an injection point with @Exact allows you to select an exact implementation of the injection point type to inject. For example:

```
interface PaymentService {

  ...

}
```

```
class ChequePaymentService implements PaymentService {

  ...

}
```

```
class CardPaymentService implements PaymentService {

  ...

}
```

```
class PaymentProcessor {

  @Inject @Exact(CardPaymentService.class)
  PaymentService paymentService;


  ...


}
```

4.3. @Client

It is common to want to qualify a bean as belonging to the current client (for example we want to differentiate the default system locale from the current client's locale). Solder provides a built in qualifier, `@Client` for this purpose.

4.4. Named packages

Solder allows you to annotate the package `@Named`, which causes every bean defined in the package to be given its default name. Package annotations are defined in the file `package-info.java`. For example, to cause any beans defined in `com.acme` to be given their default name:

```
@Named
package com.acme
```

4.5. @FullyQualified bean names

According to the CDI standard, the `@Named` annotation assigns a name to a bean equal to the value specified in the `@Named` annotation or, if a value is not provided, the simple name of the bean class. This behavior aligns with the needs of most application developers. However, framework writers should avoid trampling on the "root" bean namespace. Instead, frameworks should specify qualified names for built-in components. The motivation is the same as qualifying Java types. The `@FullyQualified` provides this facility without sacrificing type-safety.

Solder allows you to customize the bean name using the complementary `@FullyQualified` annotation. When the `@FullyQualified` annotation is added to a `@Named` bean type, producer method or producer field, the standard bean name is prefixed with the name of the Java package in which the bean resides, the segments separated by a period. The resulting fully-qualified bean name (FQBN) replaces the standard bean name.

```java
package com.acme;

@FullyQualified @Named
public class NamedBean {
  public int getAge()
  {
    return 5;
  }
}
```

The bean in the previous code listing is assigned the name `com.acme.namedBean`. The value of its property `age` would be referenced in an EL expression (perhaps in a JSF view template) as follows:

```
#{com.acme.namedBean.age}
```

The `@FullyQualified` annotation is permitted on a bean type, producer method or producer field. It can also be used on a Java package, in which case all `@Named` beans in that package get a bean name which is fully-qualified.

```java
@FullyQualified
package com.acme;
```

If you want to use a different Java package as the namespace of the bean, rather than the Java package of the bean, you specify any class in that alternative package in the annotation value.

```
package com.acme;

@FullyQualified(ClassInOtherPackage.class) @Named
public class CustomNamespacedNamedBean {
    ...
}
```

Annotation Literals

Solder provides a complete set of `AnnotationLiteral` classes corresponding to the annotation types defined in the CDI (JSR-299) and Injection (JSR-330) specifications. These literals are located in the `org.jboss.solder.literal` package.

For any annotation that does not define an attribute, its corresponding `AnnotationLiteral` contains a static `INSTANCE` member. You should use this static member whenever you need a reference to an annotation instance rather than creating a new instance explicitly.

```
new AnnotatedTypeBuilder<X>().readFromType(type).addToClass(NamedLiteral.INSTANCE);
```

Literals are provided for the following annotations from *Context and Dependency Injection* (including annotations from *Dependency Injection for Java*):

- `@Alternative`

- `@Any`

- `@ApplicationScoped`

- `@ConversationScoped`

- `@Decorator`

- `@Default`

- `@Delegate`

- `@Dependent`

- `@Disposes`

- `@Inject`

- `@Model`

- `@Named`

- `@New`

- `@Nonbinding`

- `@NormalScope`

- `@Observes`

- `@Produces`

- `@RequestScoped`

- `@SessionScoped`

- `@Specializes`

- `@Stereotype`

- `@Typed`

Literals are also provided for the following annotations from *Solder*:

- `@Client`

- `@DefaultBean`

- `@Exact`

- `@Generic`

- `@GenericType`

- `@Mapper`

- `@MessageBundle`

- `@Requires`

- `@Resolver`

- `@Resource`

- `@Unwraps`

- `@Veto`

For more information about these annotations, consult the corresponding API documentation.

Evaluating Unified EL

Solder provides a method to evaluate EL that is not dependent on JSF or JSP, a facility sadly missing in Java EE. To use it inject `Expressions` into your bean. You can evaluate value expressions, or method expressions. The Solder API provides type inference for you. For example:

```java
class FruitBowl {

  @Inject Expressions expressions;

  public void run() {
    String fruitName = expressions.evaluateValueExpression("#{fruitBowl.fruitName}");
    Apple fruit = expressions.evaluateMethodExpression("#{fruitBown.getFruit}");
  }
}
```

6.1. @Resolver

Solder also contains a qualifier to ease registration of `javax.el.ELResolver` instances. The `@Resolver` will register any `javax.el.ELResolver` annotated with `@Resolver` with the application wide `javax.el.ELResolver`.

Resource Loading

Solder provides an extensible, injectable resource loader. The resource loader can provide URLs or managed input streams. By default the resource loader will look at the classpath, and the servlet context if available.

If the resource name is known at development time, the resource can be injected, either as a URL or an InputStream:

```
@Inject
@Resource("WEB-INF/beans.xml")
URL beansXml;

@Inject
@Resource("WEB-INF/web.xml")
InputStream webXml;
```

If the resource name is not known, the `ResourceProvider` can be injected, and the resource looked up dynamically:

```
@Inject
void readXml(ResourceProvider provider, String fileName) {
    InputStream is = provider.loadResourceStream(fileName);
}
```

If you need access to all resources under a given name known to the resource loader (as opposed to first resource loaded), you can inject a collection of resources:

```
@Inject
@Resource("WEB-INF/beans.xml")
Collection<URL> beansXmls;

@Inject
@Resource("WEB-INF/web.xml")
Collection<InputStream> webXmls;
```

> **Tip**
>
> Any input stream injected, or created directly by the `ResourceProvider` is managed, and will be automatically closed when the bean declaring the injection point of the resource or provider is destroyed.

If the resource is a Properties bundle, you can also inject it as a set of `Properties`:

```
@Inject
@Resource("META-INF/aws.properties")
Properties awsProperties;
```

7.1. Extending the Resource Loader

If you want to load resources from another location, you can provide an additional resource loader. First, create the resource loader implementation:

```
class MyResourceLoader implements ResourceLoader {
  ...
}
```

And then register it as a service by placing the fully qualified class name of the implementation in a file called `META-INF/services/org.jboss.solder.resourceLoader.ResourceLoader`.

Logging, redesigned

Solder brings a fresh perspective to the ancient art of logging. Rather than just giving you an injectable version of the same old logging APIs, Solder goes the extra mile by embracing the type-safety of CDI and eliminating brittle, boilerplate logging statements. The best part is, no matter how you decide to roll it out, you still get to keep your logging engine of choice (for the logging wars will never end!).

8.1. JBoss Logging: The foundation

Before talking about Solder Logging, you need to first be introduced to JBoss Logging 3. The reason is, JBoss Logging provides the foundation on which Solder's declarative programming model for logging is built. Plus, we have to convince you that you *aren't* tied to JBoss AS by using it.

JBoss Logging acts as a logging bridge. If you don't add any other logging libraries to your project, it will delegate all logging calls it handles to the logging facility built into the Java platform (commonly referred to as JDK logging). That's nice, because it means your deployment headaches caused by missing logging jars are gone. And you accomplish it all through the use of the *Logger* [http://docs.jboss.org/seam/3/solder/latest/api/org/jboss/logging/Logger.html] type. It has the usual level-based log methods and complimentary ones that provide formatting.

Here's an example of how you obtain a logger and log a basic message:

```
Logger log = Logger.getLogger(Bean.class);
// log a plain text method
log.debug("I'm using JBoss Logging.");
```

If you want to use another logging engine, such as SLF4J or Log4J, you just have to add the native library to the deployment. Keep in mind, though, if your application server provides one of these frameworks, it will get chosen instead. On JBoss AS, JBoss Logging will prefer the JBoss LogManager because it's the built-in logging engine. (We are looking into more sophisticated runtime selection of the logging engine).

Here are the providers JBoss Logging supports (and the order in which it looks for them):

- JBoss LogManager

- Log4J

- SLF4J

- JDK logging

So you get that JBoss Logging is an abstraction. What else is it good for?

JBoss Logging has a facility for formatting log messages, using either the printf syntax or `MessageFormat`. This makes it possible to use positional parameters to build dynamic log messages based on contextual information.

```
Logger log = Logger.getLogger(Bean.class);
// log a message formatted using printf-style substitutions
log.infof("My name is %s.", "David");
// log a message formatted using MessageFormat-style substitutions
log.errorv("The license for Solder is the {0}", "APL");
```

The most significant and distinguishing feature of JBoss Logging is support for typed loggers. A typed logger is an interface that defines methods which serve as logging operations. When a method is invoked on one of these interfaces, the message defined in an annotation on the method is interpolated and written to the underlying logging engine.

Here's an example of a typed logger:

```
import org.jboss.logging.Message;
import org.jboss.logging.LogMessage;
import org.jboss.logging.MessageLogger;

@MessageLogger
public interface CelebritySightingLog {

   @LogMessage @Message("Spotted celebrity %s!")
   void spottedCelebrity(String name);

}
```

JBoss Logging has parallel support for typed message bundles, whose methods return a formatted message rather than log it. Combined, these features form the centerpiece of Solder's logging and message bundle programming model (and a foundation for additional support provided by the Seam international module). After looking at the samples provided so far, don't pull out your IDE just yet. We'll get into the details of typed loggers and how to use them in Solder in a later section.

There you have it! JBoss Logging is a low-level API that provides logging abstraction, message formatting and internationalization, and typed loggers. *But it doesn't tie you to JBoss AS!*

With that understanding, we'll now move on to what Solder does to turn this foundation into a programming model and how to use it in your CDI-based application.

8.2. Solder Logging: Feature set

Solder builds on JBoss Logging 3 to provide the following feature set:

- An abstraction over common logging backends and frameworks (such as JDK Logging, log4j and slf4j)

- Injectable loggers and message bundles

- Innovative, typed message loggers and message bundles defined using interfaces

- Build time tooling to generate typed loggers for production

- Full support for internationalization and localization:

  - Developers work with interfaces and annotations only

  - Translators work with message bundles in properties files

- Access to the "Mapped Diagnostic Context" (MDC) and/or the "Nested Diagnostic Context" (NDC) (if the underlying logger supports it)

- Serializable loggers for use in contextual components

> **Note**
>
> Seam's international module builds on this programming model to provide even more features for producing localized message strings.

Without further discussion, let's get into it.

8.3. Typed loggers

To define a typed logger, first create an interface, annotated it, then add methods that will act as log operations and configure the message it will print using another annotation:

```java
import org.jboss.solder.messages.Message;
import org.jboss.solder.logging.Log;
import org.jboss.solder.logging.MessageLogger;

@MessageLogger
public interface TrainSpotterLog {

    @Log @Message("Spotted %s diesel trains")
    void dieselTrainsSpotted(int number);
```

```
}
```

We have configured the log messages to use printf-style interpolations of parameters (%s).

> **Note**
>
> Make sure you are using the annotations from Solder (`org.jboss.solder.messages` and `org.jboss.solder.logging` packages only).

You can then inject the typed logger with no further configuration necessary. We use another optional annotation to set the category of the logger to "trains" at the injection point, overriding the default category of the fully-qualified class name of the component receiving the injection:

```
@Inject @Category("trains")
private TrainSpotterLog log;
```

We log a message by simply invoking a method of the typed logger interface:

```
log.dieselTrainsSpotted(7);
```

The default locale will be used unless overridden. Here we configure the logger to use the UK locale:

```
@Inject @Category("trains") @Locale("en_GB")
private TrainSpotterLog log;
```

You can also log exceptions.

```
import org.jboss.solder.messages.Message;
import org.jboss.solder.messages.Cause;
import org.jboss.solder.logging.Log;
import org.jboss.solder.logging.MessageLogger;

@MessageLogger
public interface TrainSpotterLog {
```

```
    @Log @Message("Failed to spot train %s")
    void missedTrain(String trainNumber, @Cause Exception exception);


}
```

You can then log a message with an exception as follows:

```
try {
    ...
} catch (Exception e) {
    log.missedTrain("RH1", e);
}
```

The stacktrace of the exception parameter will be written to the log along with the message.

Typed loggers also provide internationalization support. Simply add the `@MessageBundle` annotation to the logger interface.

If injecting a typed logger seems too "enterprisy" to you, or you need to get a reference to it from outside of CDI, you can use a static accessor method on `Logger`:

```
TrainSpotterLog log = Logger.getMessageLogger(TrainSpotterLog.class, "trains");
log.dieselTrainsSpotted(7);
```

The injected version is a convenience for those who prefer the declarative style of programming. If you are looking for a simpler starting point, you can simply use the `Logger` directly.

8.4. Native logger API

You can also inject a "plain old" Logger (from the JBoss Logging API):

```
import javax.inject.Inject;

import org.jboss.solder.logging.Logger;

public class LogService {
    @Inject
    private Logger log;

    public void logMessage() {
        log.info("Hey sysadmins!");
    }
```

```
}
```

Log messages created from this Logger will have a category (logger name) equal to the fully-qualified class name of the bean implementation class. You can specify a category explicitly using an annotation.

```
@Inject @Category("billing")
private Logger log;
```

You can also specify a category using a reference to a type:

```
@Inject @TypedCategory(BillingService.class)
private Logger log;
```

8.5. Typed message bundles

Often times you need to access a localized message. For example, you need to localize an exception message. Solder let's you retrieve this message from a typed message logger to avoid having to use hard-coded string messages.

To define a typed message bundle, first create an interface, annotated it, then add methods that will act as message retrievers and configure the message to produce using another annotation:

```
import org.jboss.solder.messages.Message;
import org.jboss.solder.messages.MessageBundle;

@MessageBundle
public interface TrainMessages {

    @Message("No trains spotted due to %s")
    String noTrainsSpotted(String cause);

}
```

Inject it:

```
@Inject @MessageBundle
private TrainMessages messages;
```

And use it:

```
throw new BadDayException(messages.noTrainsSpotted("leaves on the line"));
```

8.6. Implementation classes

You may have noticed that throughout this chapter, we've only defined interfaces. Yet, we are injecting and invoking them as though they are concrete classes. So where's the implementation?

Good news. The typed logger and message bundle implementations are generated automatically! You'll see this strategy used often in Seam 3. It's declarative programming at its finest (or to an extreme, depending on how you look at it). Either way, it saves you from a whole bunch of typing.

So *how* are they generated? Let's find out!

8.6.1. Generating the implementation classes

The first time you need logging in your application, you'll likely start with the more casual approach of using the `Logger` API directly. There's no harm in that, but it's certainly cleaner to use the typed loggers, and at the same time leverage the parallel benefits of the typed bundles. So we recommend that as your long term strategy.

Once you are ready to move to the the typed loggers and message bundles, you'll need to generate the concrete implementation classes as part of the build. These classes are generated by using an *annotation processor* that is provided by Solder and based on the *JBoss Logging tools project* [https://github.com/jamezp/jboss-logging-tools]. Don't worry, setting it up is a lot simpler than it sounds. You just need to do these two simple steps:

- Set the Java compliance to 1.6 (or better)

- Add the Solder tooling library to the build classpath

> ⚠️ **Warning**
>
> If you forget to add the annotation processor to your build, you'll get an error when you deploy the application that reports: "Invalid bundle interface (implementation not found)". This error occurs because the concrete implementation classes are missing.

Setting the Java compliance to 1.6 enables any annotation processors on the classpath to be activated during compilation.

If you're using Maven, here's how the configuration in your POM file looks:

```
<dependencies>
```

```xml
  <!-- Annotation processor for generating typed logger and message bundle classes -->
  <dependency>
    <groupId>org.jboss.solder</groupId>
    <artifactId>solder-tooling</artifactId>
    <scope>provided</scope>
    <optional>true</optional>
  </dependency>
  ...
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

> **Note**
>
> In the future, you can expect IDE plugins like JBoss Tools to setup this configuration automatically.

Here are the classes that will be generated for the examples above:

```
TrainSpotterLog_$logger.java
TrainSpotterLog_$logger_en_GB.java
TrainMessages_$bundle.java
```

Classes are generated for each language referenced by an annotation or if there is a .i18n.properties language file in the same package as the interface and has the same root name. For instance, if we wanted to generate a French version of `TrainMessages`, we would have to create the following properties file in the same package as the interface:

```
TrainMessages.i18n_fr.properties
```

Then populate it with the translations (Note the property key is the method name):

noTrainsSpotted=pas de trains repéré en raison de %s

Now the annotation processor will generate the following class:

TrainMessages_$bundle_fr.java

Now you can add typed loggers and message bundles at will (and you won't have to worry about unsatisfied dependencies).

8.6.2. Including the implementation classes in Arquillian tests

If you are writing an Arquillian test, be sure to include the concrete classes in the ShrinkWrap archive. Otherwise, you may receive an exception like:

Invalid bundle interface org.example.log.AppLog (implementation not found)

The best approach is to put your typed message loggers and bundles in their own package. Then, you include the package in the ShrinkWrap archive:

```
ShrinkWrap.create(JavaArchive.class, "test.jar")
      .addPackage(AppLog.class.getPackage());
```

This strategy will effectively package the interface and the generated implementation class(es) (even though you can't see the generated implementation classes in your source tree).

Annotation and AnnotatedType Utilities

Solder provides a number of utility classes that make working with annotations and `AnnotatedType`s easier. This chapter walks you through each utility, and gives you some ideas about how to use it. For more detail, take a look at the JavaDoc on each class.

9.1. Annotated Type Builder

Solder provides an `AnnotatedType` implementation that should be suitable for the needs of most portable extensions. The `AnnotatedType` is created from `AnnotatedTypeBuilder`, typically in an extension's observer method, as follows:

```
AnnotatedTypeBuilder builder = new AnnotatedTypeBuilder()
     .readFromType(type, true) /* readFromType can read from an AnnotatedType or a class */
     .addToClass(ModelLiteral.INSTANCE); /* add the @Model annotation */
     .create()
```

Here we create a new builder, and initialize it using an existing `AnnotatedType`. We can then add or remove annotations from the class, and its members. When we have finished modifying the type, we call `create()` to spit out a new, immutable, `AnnotatedType`.

```
AnnotatedType redefinedType = builder.create();
```

One place this is immensely useful is for replacing the `AnnotatedType` in an extension that observes the `ProcessAnnotatedType` event:

```
public <X> void processAnnotatedType(@Observes ProcessAnnotatedType<X> evt) {
   AnnotatedTypeBuilder builder = new AnnotatedTypeBuilder()
       .readFromType(evt.getAnnotatedType(), true)
       .addToClass(ModelLiteral.INSTANCE);
   evt.setAnnotatedType(builder.create());
}
```

This type is now effectively annotated with `@Model`, even if the annotation is not present on the class definition in the Java source file.

`AnnotatedTypeBuilder` also allows you to specify a "redefinition", which can be applied to the type, a type of member, or all members. The redefiner will receive a callback for any annotations present which match the annotation type for which the redefinition is applied.

For example, to remove the qualifier `@Unique` from the type and any of its members, use this:

```
AnnotatedTypeBuilder builder = new AnnotatedTypeBuilder()
    .readFromType(type, true)
    .redefine(Unique.class, new AnnotationRedefiner<Unique>() {

      public void redefine(RedefinitionContext<Unqiue> ctx) {
        ctx.getAnnotationBuilder().remove(Unique.class);
      }
    });
AnnotatedType redefinedType = builder.create();
```

No doubt, this is a key blade in Solder's army knife arsenal of tools. You can quite effectively change the picture of the type metadata CDI discovers when it scans and processes the classpath of a bean archive.

9.2. Annotation Instance Provider

Sometimes you may need an annotation instance for an annotation whose type is not known at development time. Solder provides a `AnnotationInstanceProvider` class that can create an `AnnotationLiteral` instance for any annotation at runtime. Annotation attributes are passed in via a `Map<String,Object>`. For example given the follow annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MultipleMembers {
    int intMember();

    long longMember();

    short shortMember();

    float floatMember();

    double doubleMember();

    byte byteMember();

    char charMember();
```

```
    boolean booleanMember();


    int[] intArrayMember();
}
```

We can create an annotation instance as follows:

```
/* Create a new provider */
AnnotationInstanceProvider provider = new AnnotationInstanceProvider();

/* Set the value for each of attributes */
Map<String, Object> values = new HashMap<String, Object>();
values.put("intMember", 1);
values.put("longMember", 1);
values.put("shortMember", 1);
values.put("floatMember", 0);
values.put("doubleMember", 0);
values.put("byteMember", ((byte) 1));
values.put("charMember", 'c');
values.put("booleanMember", true);
values.put("intArrayMember", new int[] { 0, 1 });


/* Generate the instance */
MultipleMembers an = provider.get(MultipleMembers.class, values);
```

9.3. Annotation Inspector

The Annotation Inspector allows you to easily discover annotations which are meta-annotated.
For example:

```
/* Discover all annotations on type which are meta-annotated @Constraint */
Set<Annotation> constraints = AnnotationInspector.getAnnotations(type, Constraint.class);

/* Load the annotation instance for @FacesValidator the annotation may declared on the type, */
/* or, if the type has any stereotypes, on the stereotypes */
FacesValidator validator = AnnotationInspector.getAnnotation(
     type, FacesValidator.class, true, beanManager);
```

The utility methods work correctly on `Stereotype`s as well. Let's say you're working with a bean that was decorated `@Model`, running the following example will still show you the underlying `@Named`

```
// assuming you have a class..
@Model
public class User {

    ...

}

// Assume type represents the User class
assert AnnotationInspector.isAnnotationPresent(type, Named.class, beanManager);

// Retrieves the underlying @Named instance on the stereotype
Named name = AnnotationInspector.getAnnotation(type, Named.class, true, beanManager);
```

The search algorithm will first check to see if the annotation is present directly on the annotated element first, then searches within the stereotype annotations on the element. If you only want to search for `Annotation`s on `Stereotype`s, then you can use either of the methods `AnnotationInspector.getAnnotationFromStereotype`.

There is an overloaded form of `isAnnotationPresent` and `getAnnotation` to control whether it will search on `Stereotype`s or not. For both of these methods, a search is performed first directly on the element before searching in stereotypes.

9.4. Synthetic Qualifiers

When developing an extension to CDI, it can be useful to detect certain injection points, or bean definitions and based on annotations or other metadata, add qualifiers to further disambiguate the injection point or bean definition for the CDI bean resolver. Solder's synthetic qualifiers can be used to easily generate and track such qualifiers.

In this example, we will create a synthetic qualifier provider, and use it to create a qualifier. The provider will track the qualifier, and if a qualifier is requested again for the same original annotation, the same instance will be returned.

```
/* Create a provider, giving it a unique namespace */
Synthetic.Provider provider = new Synthetic.Provider("com.acme");

/* Get the a synthetic qualifier for the original annotation instance */
Synthetic synthetic = provider.get(originalAnnotation);

/* Later calls with the same original annotation instance will return the same instance */
/* Alternatively, we can "get and forget" */

Synthetic synthetic2 = provider.get();
```

9.5. Reflection Utilities

Solder comes with a number miscellaneous reflection utilities; these extend JDK reflection, and some also work on CDI's Annotated metadata. See the javadoc on `Reflections` for more.

Solder also includes a simple utility, `PrimitiveTypes` for converting between primitive and their respective wrapper types, which may be useful when performing data type conversion. Sadly, this is functionality which is missing from the JDK.

`InjectableMethod` allows an `AnnotatedMethod` to be injected with parameter values obtained by following the CDI type safe resolution rules, as well as allowing the default parameter values to be overridden.

Obtaining a reference to the BeanManager

When developing a framework that builds on CDI, you may need to obtain the `BeanManager` for the application, you can't simply inject it as you are not working in an object managed by the container. The CDI specification allows lookup of `java:comp/BeanManager` in JNDI, however, some environments don't support binding to this location (e.g. servlet containers such as Tomcat and Jetty) and some environments don't support JNDI (e.g. the Weld SE container). For this reason, most framework developers will prefer to avoid a direct JNDI lookup.

Often it is possible to pass the correct `BeanManager` to the object in which you require it, for example via a context object. For example, you might be able to place the `BeanManager` in the `ServletContext`, and retrieve it at a later date.

On some occasions however there is no suitable context to use, and in this case, you can take advantage of the abstraction over `BeanManager` lookup provided by Solder. To lookup up a `BeanManager`, you can extend the abstract `BeanManagerAware` class, and call `getBeanManager()`:

```
public class WicketIntegration extends BeanManagerAware {

  public WicketManager getWicketManager() {
    Bean<?> bean = getBeanManager().getBeans(IRequestListener.class);
    ... // and so on to lookup the bean
  }

}
```

The benefit here is that `BeanManagerAware` class will first look to see if its `BeanManager` injection point was satisfied before consulting the providers. Thus, if injection becomes available to the class in the future, it will automatically start the more efficient approach.

Occasionally you will be working in an existing class hierarchy, in which case you can use the accessor on `BeanManagerLocator`. For example:

```
public class ResourceServlet extends HttpServlet {

  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
    BeanManager beanManager = new BeanManagerLocator().getBeanManager();
    ...
```

```
  }
}
```

If this lookup fails to resolve a `BeanManager`, the `BeanManagerUnavailableException`, a runtime exception, will be thrown. If you want to perform conditional logic based on whether the `BeanManager` is available, you can use this check:

```java
public class ResourceServlet extends HttpServlet {

  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
    BeanManagerLocator locator = new BeanManagerLocator();
    if (locator.isBeanManagerAvailable()) {
      BeanManager beanManager = locator.getBeanManager();
      ... // work with the BeanManager
    }
    else {
      ... // work without the BeanManager
    }
  }
}
```

However, keep in mind that you can inject into Servlets in Java EE 6!! So it's very likely the lookup isn't necessary, and you can just do this:

```java
public class ResourceServlet extends HttpServlet {

  @Inject
  private BeanManager beanManager;

  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
    ... // work with the BeanManager
  }
}
```

Bean Utilities

Solder provides a number of base classes which can be extended to create custom beans. Solder also provides bean builders which can be used to dynamically create beans using a fluent API.

AbstractImmutableBean

> An immutable (and hence thread-safe) bean, whose constructor will substitute specification defaults if `null` is passed for a particular attribute. Subclasses must implement the `create()` and `destroy()` methods.

AbstractImmutableProducer

> An immutable (and hence thread-safe) abstract class for creating producers. Subclasses must implement `produce()` and `dispose()`.

BeanBuilder

> A builder for creating immutable beans which can read the type and annotations from an `AnnotatedType`.

Beans

> A set of utilities for working with beans.

ForwardingBean

> A base class for implementing `Bean` which forwards all calls to `delegate()`.

ForwardingInjectionTarget

> A base class for implementing `InjectionTarget` which forwards all calls to `delegate()`.

ForwardingObserverMethod

> A base class for implementing `ObserverMethod` which forwards all calls to `delegate()`.

ImmutableBean

> An immutable (and hence thread-safe) bean, whose constructor will substitute specification defaults if `null` is passed for a particular attribute. An implementation of `ContextualLifecycle` may be registered to receive lifecycle callbacks.

ImmutableInjectionPoint

> An immutable (and hence thread-safe) injection point.

ImmutableNarrowingBean

> An immutable (and hence thread-safe) narrowing bean. Narrowing beans allow you to build a general purpose bean (likely a producer method), and register it for a narrowed type (or qualifiers).

ImmutablePassivationCapableBean

> An immutable (and hence thread-safe) bean, whose constructor will substitute specification defaults if `null` is passed for a particular attribute. An implementation of

`ContextualLifecycle` may be registered to receive lifecycle callbacks. The bean implements `PassivationCapable`, and an id must be provided.

`ImmutablePassivationCapableNarrowingBean`

An immutable (and hence thread-safe) narrowing bean. Narrowing beans allow you to build a general purpose bean (likely a producer method), and register it for a narrowed type (or qualifiers). The bean implements `PassivationCapable`, and an id must be provided.

`NarrowingBeanBuilder`

A builder for creating immutable narrowing beans which can read the type and annotations from an `AnnotatedType`.

The use of these classes is in general trivially understood with an understanding of basic programming patterns and the CDI specification, so no in depth explanation is provided here. The JavaDoc for each class and method provides more detail.

Properties

Properties are a convenient way of locating and working with *JavaBean* [http://en.wikipedia.org/wiki/JavaBean] properties. They can be used with properties exposed via a getter/setter method, or directly via the field of a bean, providing a uniform interface that allows you all properties in the same way.

Property queries allow you to interrogate a class for properties which match certain criteria.

12.1. Working with properties

The `Property<V>` interface declares a number of methods for interacting with bean properties. You can use these methods to read or set the property value, and read the property type information. Properties may be readonly.

Table 12.1. Property methods

Method	Description	
`String getName();`	Returns the name of the property.	
`Type getBaseType();`	Returns the property type.	
`Class<V> getJavaClass();`	Returns the property class.	
`AnnotatedElement getAnnotatedElement();`	Returns the annotated element -either the `Field` or `Method` that the property is based on.	
`V getValue();`	Returns the value of the property.	
`void setValue(V value);`	Sets the value of the property.	
`Class<?> getDeclaringClass();`	Gets the class declaring the property.	
`boolean isReadOnly();`	Check if the property can be written as well as read.	
`Member getMember();`	Get the class member which retrieves the property (i.e. field or getter).	
`void setAccessible`	Sets the `Member` to be accessible to changes. Should be performed within a `PriviledgedAction` to work correctly with Security Managers.	

Given a class with two properties, `personName` and `postcode`:'

```java
class Person {

  PersonName personName;

  Address address;

  void setPostcode(String postcode) {
    address.setPostcode(postcode);
  }

  String getPostcode() {
    return address.getPostcode();
  }

}
```

You can create two properties:

```java
Property<PersonName> personNameProperty = Properties.createProperty(Person.class.getField("personName")
Property<String> postcodeProperty = Properties.createProperty(Person.class.getMethod("getPostcode"));
```

12.2. Querying for properties

To create a property query, use the `PropertyQueries` class to create a new `PropertyQuery`
instance:

```java
PropertyQuery<?> query = PropertyQueries.createQuery(Foo.class);
```

If you know the type of the property that you are querying for, you can specify it via a type
parameter:

```java
PropertyQuery<String> query = PropertyQueries.<String>createQuery(identityClass);
```

12.3. Property Criteria

Once you have created the `PropertyQuery` instance, you can add search criteria. Solder provides three built-in criteria types, and it is very easy to add your own. A criteria is added to a query via the `addCriteria()` method. This method returns an instance of the `PropertyQuery`, so multiple `addCriteria()` invocations can be stacked.

12.3.1. AnnotatedPropertyCriteria

This criteria is used to locate bean properties that are annotated with a certain annotation type. For example, take the following class:

```java
public class Foo {
    private String accountNumber;
    private @Scrambled String accountPassword;
    private String accountName;
}
```

To query for properties of this bean annotated with `@Scrambled`, you can use an `AnnotatedPropertyCriteria`, like so:

```java
PropertyQuery<String> query = PropertyQueries.<String>createQuery(Foo.class)
    .addCriteria(new AnnotatedPropertyCriteria(Scrambled.class));
```

This query matches the `accountPassword` property of the `Foo` bean.

12.3.2. NamedPropertyCriteria

This criteria is used to locate a bean property with a particular name. Take the following class:

```java
public class Foo {
    public String getBar() {
        return "foobar";
    }
}
```

The following query will locate properties with a name of `"bar"`:

```java
PropertyQuery<String> query = PropertyQueries.<String>createQuery(Foo.class)
```

```
        .addCriteria(new NamedPropertyCriteria("bar"));
```

12.3.3. TypedPropertyCriteria

This criteria can be used to locate bean properties with a particular type.

```
public class Foo {
  private Bar bar;
}
```

The following query will locate properties with a type of `Bar`:

```
PropertyQuery<Bar> query = PropertyQueries.<Bar>createQuery(Foo.class)
    .addCriteria(new TypedPropertyCriteria(Bar.class));
```

12.3.4. Creating a custom property criteria

To create your own property criteria, simply implement the `org.jboss.solder.properties.query.PropertyCriteria` interface, which declares the two methods `fieldMatches()` and `methodMatches`. In the following example, our custom criteria implementation can be used to locate whole number properties:

```
public class WholeNumberPropertyCriteria implements PropertyCriteria {
  public boolean fieldMatches(Field f) {
    return f.getType() == Integer.class || f.getType() == Integer.TYPE.getClass() ||
        f.getType() == Long.class || f.getType() == Long.TYPE.getClass() ||
        f.getType() == BigInteger.class;
  }

  public boolean methodMatches(Method m) {
    return m.getReturnType() == Integer.class || m.getReturnType() == Integer.TYPE.getClass() ||
        m.getReturnType() == Long.class || m.getReturnType() == Long.TYPE.getClass() ||
        m.getReturnType() == BigInteger.class;
  }
}
```

12.4. Fetching the results

After creating the `PropertyQuery` and setting the criteria, the query can be executed by invoking either the `getResultList()` or `getFirstResult()` methods. The `getResultList()` method

returns a `List` of `Property` objects, one for each matching property found that matches all the specified criteria:

```
List<Property<String>> results = PropertyQueries.<String>createQuery(Foo.class)
  .addCriteria(new TypedPropertyCriteria(String.class))
  .getResultList();
```

If no matching properties are found, `getResultList()` will return an empty `List`. If you know that the query will return exactly one result, you can use the `getFirstResult()` method instead:

```
Property<String> result = PropertyQueries.<String>createQuery(Foo.class)
  .addCriteria(new NamedPropertyCriteria("bar"))
  .getFirstResult();
```

If no properties are found, then `getFirstResult()` will return null. Alternatively, if more than one result is found, then `getFirstResult()` will return the first property found.

Alternatively, if you know that the query will return exactly one result, and you want to assert that assumption is true, you can use the `getSingleResult()` method instead:

```
Property<String> result = PropertyQueries.<String>createQuery(Foo.class)
  .addCriteria(new NamedPropertyCriteria("bar"))
  .getSingleResult();
```

If no properties are found, or more than one property is found, then `getSingleResult()` will throw an exception. Otherwise, `getSingleResult()` will return the sole property found.

Sometimes you may not be interested in read only properties, so `getResultList()`,`getFirstResult()` and `getSingleResult()` have corresponding `getWritableResultList()`,`getWritableFirstResult()` and `getWritableSingleResult()` methods, that will only return properties that are not read-only. This means that if there is a field and a getter method that resolve to the same property, instead of getting a read-only `MethodProperty` you will get a writable `FieldProperty`.

Unwrapping Producer Methods

Unwrapping producer methods allow you to create injectable objects that have "self-managed" lifecycles, and are particularly useful if you have need a bean whose lifecycle does not exactly match one of the lifecycles of one of the existing scopes. The lifecycle of the bean is managed by the bean that defines the producer method, and changes to the unwrapped object are immediately visible to all clients.

You can declare a method to be an unwrapping producer method by annotating it @Unwraps. The return type of the managed producer must be proxyable (see *Section 5.4.1 of the CDI specification, "Unproxyable bean types"* [http://docs.jboss.org/cdi/spec/1.0/html/injectionelresolution.html#unproxyable]). Every time a method is called on unwrapped object the invocation is forwarded to the result of calling the unwrapping producer method - the unwrapped object.

> **Important**
>
> Solder implements this by injecting a proxy rather than the original object. Every invocation on the injected proxy will cause the unwrapping producer method to be invoked to obtain the instance on which to invoke the method called. Solder will then invoke the method on unwrapped instance.
>
> Because of this, it is very important the producer method is lightweight.

For example consider a permission manager (that manages the current permission), and a security manager (that checks the current permission level). Any changes to permission in the permission manager are immediately visible to the security manager.

```
@SessionScoped
class PermissionManager {

  Permission permission;

  void setPermission(Permission permission) {
    this.permission=permission;
  }

  @Unwraps @Current
  Permission getPermission() {
    return this.permission;
  }
}
```

```
@SessionScoped
class SecurityManager {

  @Inject @Current
  Permission permission;

  boolean checkAdminPermission() {
    return permission.getName().equals("admin");
  }

}
```

When `permission.getName()` is called, the unwrapped Permission forwards the invocation of `getName()` to the result of calling `PermissionManager.getPermission()`.

For example you could raise the permission level before performing a sensitive operation, and then lower it again afterwards:

```
public class SomeSensitiveOperation {

  @Inject
  PermissionManager permissionManager;

  public void perform() {
    try {
      permissionManager.setPermission(Permissions.ADMIN);
      // Do some sensitive operation
    } finally {
      permissionManager.setPermission(Permissions.USER);
    }
  }

}
```

Unwrapping producer methods can have parameters injected, including `InjectionPoint` (which represents) the calling method.

Default Beans

Suppose you have a situation where you want to provide a default implementation of a particular service and allow the user to override it as needed. Although this may sound like a job for an alternative, they have some restrictions that may make them undesirable in this situation. If you were to use an alternative it would require an entry in every `beans.xml` file in an application.

Developers consuming the extension will have to open up the any jar file which references the default bean, and edit the `beans.xml` file within, in order to override the service. This is where default beans come in.

Default beans allow you to create a default bean with a specified type and set of qualifiers. If no other bean is installed that has the same type and qualifiers, then the default bean will be installed.

Let's take a real world example - a module that allows you to evaluate EL (something that Solder provides!). If JSF is available we want to use the `FunctionMapper` provided by the JSF implementation to resolve functions, otherwise we just want to use a a default `FunctionMapper` implementation that does nothing. We can achieve this as follows:

```java
@DefaultBean(FunctionMapper.class)
@Mapper
class FunctionMapperImpl extends FunctionMapper {

  @Override
  public Method resolveFunction(String prefix, String localName) {
    return null;
  }

}
```

And in the JSF module:

```java
class FunctionMapperProvider {

  @Produces
  @Mapper
  FunctionMapper produceFunctionMapper() {
    return FacesContext.getCurrentInstance().getELContext().getFunctionMapper();
  }
}
```

If `FunctionMapperProvider` is present then it will be used by default, otherwise the default `FunctionMapperImpl` is used.

A producer method or producer field may be defined to be a default producer by placing the `@DefaultBean` annotation on the producer. For example:

```
class CacheManager {

  @DefaultBean(Cache.class)
  Cache getCache() {
    ...
  }

}
```

Any producer methods or producer fields declared on a default managed bean are automatically registered as default producers, with `Method.getGenericReturnType()` or `Field.getGenericType()` determining the type of the default producer. The default producer type can be overridden by specifying `@DefaultBean` on the producer method or field.

Generic Beans

Many common services and API's require the use of more than just one class. When exposing these services via CDI, it would be time consuming and error prone to force the end developer to provide producers for all the different classes required. Generic beans provide a solution, allowing a framework author to provide a set of related beans, one for each single configuration point defined by the end developer. The configuration points specifies the qualifiers which are inherited by all beans in the set.

To illustrate the use of generic beans, we'll use the following example. Imagine we are writing an extension to integrate our custom messaging solution "ACME Messaging" with CDI. The ACME Messaging API for sending messages consists of several interfaces:

`MessageQueue`
> The message queue, onto which messages can be placed, and acted upon by ACME Messaging

`MessageDispatcher`
> The dispatcher, responsible for placing messages created by the user onto the queue

`DispatcherPolicy`
> The dispatcher policy, which can be used to tweak the dispatch policy by the client

`MessageSystemConfiguration`
> The messaging system configuration

We want to be able to create as many `MessageQueue` configurations as they need, however we do not want to have to declare each producer and the associated plumbing for every queue. Generic beans are an ideal solution to this problem.

15.1. Using generic beans

Before we take a look at creating generic beans, let's see how we will use them.

Generic beans are configured via producer methods and fields. We want to create two queues to interact with ACME Messaging, a default queue that is installed with qualifier `@Default` and a durable queue that has qualifier `@Durable`:

```
class MyMessageQueues {

  @Produces
  @ACMEQueue("defaultQueue")
  MessageSystemConfiguration defaultQueue = new MessageSystemConfiguration();

  @Produces @Durable @ConversationScoped
```

```
  @ACMEQueue("durableQueue")
  MessageSystemConfiguration producerDefaultQueue() {
    MessageSystemConfiguration config = new MessageSystemConfiguration();
    config.setDurable(true);
    return config;
  }
}
```

Looking first at the default queue, in addition to the `@Produces` annotation, the generic configuration annotation `ACMEQueue`, is used, which defines this to be a generic configuration point for ACME messaging (and cause a whole set of beans to be created, exposing for example the dispatcher). The generic configuration annotation specifies the queue name, and the value of the producer field defines the messaging system's configuration (in this case we use all the defaults). As no qualifier is placed on the definition, `@Default` qualifier is inherited by all beans in the set.

The durable queue is defined as a producer method (as we want to alter the configuration of the queue before having Solder use it). Additionally, it specifies that the generic beans created (that allow for their scope to be overridden) should be placed in the conversation scope. Finally, it specifies that the generic beans created should inherit the qualifier `@Durable`.

We can now inject our generic beans as normal, using the qualifiers specified on the configuration point:

```
class MessageLogger {

  @Inject
  MessageDispatcher dispatcher;

  void logMessage(Payload payload) {
    /* Add metaddata to the message */
    Collection<Header> headers = new ArrayList<Header>();
    ...
    Message message = new Message(headers, payload);
    dispatcher.send(message);
  }

}
```

```
class DurableMessageLogger {

  @Inject @Durable
  MessageDispatcher dispatcher;
```

```
  @Inject @Durable
  DispatcherPolicy policy;


  /* Tweak the dispatch policy to enable duplicate removal */
  @Inject
  void tweakPolicy(@Durable DispatcherPolicy policy) {
    policy.removeDuplicates();
  }

  void logMessage(Payload payload) {
    ...
  }
}
```

It is also possible to configure generic beans using beans by sub-classing the configuration type, or installing another bean of the configuration type through the SPI (e.g. using Solder Config). For example to configure a durable queue via sub-classing:

```
@Durable @ConversationScoped
@ACMEQueue("durableQueue")
class DurableQueueConfiguration extends MessageSystemConfiguration {

  public DurableQueueConfiguration()
  {
    this.durable = true;
  }
}
```

And the same thing via Solder Config:

```
<my:MessageSystemConfiguration>
  <my:Durable/>
  <s:ConversationScoped/>
  <my:ACMEQueue>durableQueue</my:ACMEQueue>
  <my:durable>true</my:durable>
</my:MessageSystemConfiguration>
```

15.2. Defining Generic Beans

Having seen how we use the generic beans, let's look at how to define them. We start by creating the generic configuration annotation:

```
@Retention(RUNTIME)
@GenericType(MessageSystemConfiguration.class)
@interface ACMEQueue {

  String value();

}
```

The generic configuration annotation a defines the generic configuration type (in this case `MessageSystemConfiguration`); the type produced by the generic configuration point must be of this type. Additionally it defines the member `name`, used to provide the queue name.

Next, we define the queue manager bean. The manager has one producer method, which creates the queue from the configuration:

```
@GenericConfiguration(ACMEQueue.class) @ApplyScope
class QueueManager {

  @Inject @Generic
  MessageSystemConfiguration systemConfig;

  @Inject
  ACMEQueue config;

  MessageQueueFactory factory;

  @PostConstruct
  void init() {
    factory = systemConfig.createMessageQueueFactory();
  }

  @Produces @ApplyScope
  public MessageQueue messageQueueProducer() {
    return factory.createMessageQueue(config.name());
  }
}
```

The bean is declared to be a generic bean for the `@ACMEQueue` generic configuration type annotation by placing the `@GenericConfiguration` annotation on the class. We can inject the generic configuration type using the `@Generic` qualifier, as well the annotation used to define the queue.

Placing the `@ApplyScope` annotation on the bean causes it to inherit the scope from the generic configuration point. As creating the queue factory is a heavy operation we don't want to do it more than necessary.

Having created the `MessageQueueFactory`, we can then expose the queue, obtaining its name from the generic configuration annotation. Additionally, we define the scope of the producer method to be inherited from the generic configuration point by placing the annotation `@ApplyScope` on the producer method. The producer method automatically inherits the qualifiers specified by the generic configuration point.

Finally we define the message manager, which exposes the message dispatcher, as well as allowing the client to inject an object which exposes the policy the dispatcher will use when queuing messages. The client can then tweak the policy should they wish.

```
@Generic
class MessageManager {

  @Inject @Generic
  MessageQueue queue;

  @Produces @ApplyScope
  MessageDispatcher messageDispatcherProducer() {
    return queue.createMessageDispatcher();
  }

  @Produces
  DispatcherPolicy getPolicy() {
    return queue.getDispatcherPolicy();
  }

}
```

Service Handler

The service handler facility allow you to declare interfaces and abstract classes as automatically implemented beans. Any call to an abstract method on the interface or abstract class will be forwarded to the invocation handler for processing.

If you wish to convert some non-type-safe lookup to a type-safe lookup, then service handlers may be useful for you, as they allow the end user to map a lookup to a method using domain specific annotations.

We will work through using this facility, taking the example of a service which can execute JPA queries upon abstract method calls. First we define the annotation used to mark interfaces as automatically implemented beans. We meta-annotate it, defining the invocation handler to use:

```java
@ServiceHandlerType(QueryHandler.class)
@Retention(RUNTIME)
@Target({TYPE})
@interface QueryService {}
```

We now define an annotation which provides the query to execute:

```java
@Retention(RUNTIME)
@Target({METHOD})
@interface Query {

  String value();

}
```

And finally, the invocation handler, which simply takes the query, and executes it using JPA, returning the result:

```java
class QueryHandler {

  @Inject EntityManager em;

  @AroundInvoke
  Object handle(InvocationContext ctx) {
    return em.createQuery(ctx.getMethod().getAnnotation(Query.class).value()).getResultList();
  }
```

```
}
```

> **ℹ Note**
>
> - The invocation handler is similar to an interceptor. It must have an `@AroundInvoke` method that returns and object and takes an `InvocationContext` as an argument.
>
> - Do not call `InvocationContext.proceed()` as there is no method to proceed to.
>
> - Injection is available into the handler class, however the handler is not a bean definition, so observer methods, producer fields and producer methods defined on the handler will not be registered.

Finally, we can define (any number of) interfaces which define our queries:

```java
@QueryService
interface UserQuery {

  @Query("select u from User u")
  public List<User> getAllUsers();
}
```

Finally, we can inject the query interface, and call methods, automatically executing the JPA query.

```java
class UserListManager {
  @Inject
  UserQuery userQuery;

  List<User> users;

  @PostConstruct
  void create() {
    users=userQuery.getAllUsers();
  }

}
```

XML Configuration Introduction

Solder provides a method for configuring CDI beans using alternate metadata sources, such as XML configuration. Currently, the XML provider is the only alternative available. Using a "type-safe" XML syntax, it is possible to add new beans, override existing beans, and add extra configuration to existing beans.

17.1. Getting Started

To take advantage of XML Configuration, you need metadata sources in the form of XML files. By default these are discovered from the classpath in the following locations:

- `/META-INF/beans.xml`

- `/META-INF/seam-beans.xml`

The `beans.xml` file is the preferred way of configuring beans via XML; however some CDI implementations will not allow this, so `seam-beans.xml` is provided as an alternative.

Here is a simple example. The following class represents a report:

```java
class Report {
 String filename;

 @Inject
 Datasource datasource;

 //getters and setters
}
```

And the following support classes:

```java
interface Datasource {
 public Data getData();
}

@SalesQualifier
class SalesDatasource implements Datasource {
 public Data getData()
 {
  //return sales data
 }
}
```

```
class BillingDatasource implements Datasource {
 public Data getData()
 {
  //return billing data
 }
}
```

The `Report` bean is fairly simple. It has a filename that tells the report engine where to load the report definition from, and a datasource that provides the data used to fill the report. We are going to configure up multiple `Report` beans via xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:java:ee"                                     ①
    xmlns:r="urn:java:org.example.reports">                   ②

 <r:Report>                                                   ③
 <s:modifies/>                                                ④
 <r:filename>sales.jrxml<r:filename>                          ⑤
 <r:datasource>
  <r:SalesQualifier/>                                         ⑥
 </r:datasource>
 </r:Report>

 <r:Report filename="billing.jrxml">                          ⑦
 <s:replaces/>                                                ⑧
 <r:datasource>
  <s:Inject/>                                                 ⑨
  <s:Exact>org.example.reports.BillingDatasource</s:Exact>    ⑩
 </r:datasource>
 </r:Report>
</beans>
```

① The namespace `urn:java:ee` is the XML Config's root namespace. This is where the built-in elements and CDI annotations live.

② There are now multiple namespaces in the `beans.xml` file. These namespaces correspond to java package names.

The namespace `urn:java:org.example.reports` corresponds to the package `org.example.reports`, where the reporting classes live. Multiple java packages can be aggregated into a single namespace declaration by separating the package names with colons, e.g. `urn:java:org.example.reports:org.example.model`. The namespaces are searched in the order they are specified in the xml document, so if two packages in the namespace have a class with the same name, the first one listed will be resolved. For more information see *Namespaces*.

③ The `<Report>` declaration configures an instance of the `Report` class as a bean.

④ Beans installed using `<s:modifies>` read annotations from the existing class, and merge them with the annotations defined via xml. In addition, if a bean is installed with `<s:modifies>`, it prevents the original class being installed as a bean. It is also possible to add new beans and replace beans altogether. For more information see *Adding, modifying and replacing beans*.

⑤ The `<r:filename>` element sets the initial value of the filename field. For more information on how methods and fields are resolved see *Configuring Methods*, and *Configuring Fields*.

⑥ The `<r:SalesQualifier>` element applies the `@SalesQualifier` to the `datasource` field. As the field already has an `@Inject` on the class definition this will cause the `SalesDatasource` bean to be injected.

⑦ This is the shorthand syntax for setting a field value.

⑧ Beans installed using `<s:replaces>` do not read annotations from the existing class. In addition, if a bean is installed with `<s:replaces>` it prevents the original class being installed as a bean.

⑨ The `<s:Inject>` element is needed as this bean was installed with `<s:replaces>`, so annotations are not read from the class definition.

⑩ The `<s:Exact>` annotation restricts the type of bean that is available for injection without using qualifiers. In this case `BillingDatasource` will be injected. This is provided as part of weld-extensions.

17.2. The Princess Rescue Example

The princess rescue example is a sample web app that uses XML Config. Run it with the following command:

```
mvn -Pjetty jetty:run
```

And then navigate to `http://localhost:9090/princess-rescue`. The XML configuration for the example is in `src/main/resources/META-INF/seam-beans.xml`.

Solder Config XML provider

18.1. XML Namespaces

The main namespace is `urn:java:ee`. This namespace contains built-in tags and types from core packages. The built-in tags are:

- `Beans`

- `modifies`

- `replaces`

- `parameters`

- `value`

- `key`

- `entry`

- `e` (alias for entry)

- `v` (alias for value)

- `k` (alias for key)

- `array`

- `int`

- `short`

- `long`

- `byte`

- `char`

- `double`

- `float`

- `boolean`

as well as classes from the following packages:

- `java.lang`

- `java.util`

- `javax.annotation`

- `javax.inject`

- `javax.enterprise.inject`

- `javax.enterprise.context`

- `javax.enterprise.event`

- `javax.decorator`

- `javax.interceptor`

- `org.jboss.solder.core`

- `org.jboss.solder.unwraps`

- `org.jboss.solder.resourceLoader`

Other namespaces are specified using the following syntax:

xmlns:my="urn:java:com.mydomain.package1:com.mydomain.package2"

This maps the namespace `my` to the packages `com.mydomain.package1` and `com.mydomain.package2`. These packages are searched in order to resolve elements in this namespace.

For example, you have a class `com.mydomain.package2.Report`. To configure a `Report` bean you would use `<my:Report>`. Methods and fields on the bean are resolved from the same namespace as the bean itself. It is possible to distinguish between overloaded methods by specifying the parameter types, for more information see *Configuring Methods*.

18.2. Adding, replacing and modifying beans

By default configuring a bean via XML creates a new bean; however there may be cases where you want to modify an existing bean rather than adding a new one. The `<s:replaces>` and `<s:modifies>` tags allow you to do this.

The `<s:replaces>` tag prevents the existing bean from being installed, and registers a new one with the given configuration. The `<s:modifies>` tag does the same, except that it merges the annotations on the bean with the annotations defined in XML. Where the same annotation is specified on both the class and in XML the annotation in XML takes precedence. This has almost

the same effect as modifying an existing bean, except it is possible to install multiple beans that modify the same class.

> **Note**
>
> Config ignores beans that have the `@Veto` annotation when using `<replaces>` and `<modifies>`.

```
<my:Report>
 <s:modifies>
 <my:NewQualifier/>
</my:Report>

<my:ReportDatasource>
 <s:replaces>
 <my:NewQualifier/>
</my:ReportDatasource>
```

The first entry above adds a new bean with an extra qualifier, in addition to the qualifiers already present, and prevents the existing `Report` bean from being installed.

The second prevents the existing bean from being installed, and registers a new bean with a single qualifier.

18.3. Applying annotations using XML

Annotations are resolved in the same way as normal classes. Conceptually, annotations are applied to the object their parent element resolves to. It is possible to set the value of annotation members using the xml attribute that corresponds to the member name. For example:

```
public @interface OtherQualifier {
    String value1();
    int value2();
    QualifierEnum value();
}
```

```
<test:QualifiedBean1>
     <test:OtherQualifier value1="AA" value2="1">A</my:OtherQualifier>
</my:QualifiedBean1>
```

```
<test:QualifiedBean2>
     <test:OtherQualifier value1="BB" value2="2" value="B" />
</my:QualifiedBean2>
```

The `value` member can be set using the inner text of the node, as seen in the first example. Type conversion is performed automatically.

> **Note**
>
> It is currently not possible set array annotation members.

18.4. Configuring Fields

It is possible to both apply qualifiers to and set the initial value of a field. Fields reside in the same namespace as the declaring bean, and the element name must exactly match the field name. For example if we have the following class:

```
class RobotFactory {
  Robot robot;
}
```

The following xml will add the `@Produces` annotation to the `robot` field:

```
<my:RobotFactory>
  <my:robot>
    <s:Produces/>
  </my:robot>
</my:RobotFactory/>
```

18.4.1. Initial Field Values

Initial field values can be set three different ways as shown below:

```
<r:MyBean company="Red Hat Inc" />

<r:MyBean>
  <r:company>Red Hat Inc</r:company>
</r:MyBean>
```

```
<r:MyBean>
 <r:company>
  <s:value>Red Hat Inc<s:value>
  <r:SomeQualifier/>
  </r:company>
</r:MyBean>
```

The third form is the only one that also allows you to add annotations such as qualifiers to the field.

It is possible to set `Map`, `Array` and `Collection` field values. Some examples:

```
<my:ArrayFieldValue>

  <my:intArrayField>
    <s:value>1</s:value>
    <s:value>2</s:value>
  </my:intArrayField>

  <my:classArrayField>
    <s:value>java.lang.Integer</s:value>
    <s:value>java.lang.Long</s:value>
  </my:classArrayField>

  <my:stringArrayField>
    <s:value>hello</s:value>
    <s:value>world</s:value>
  </my:stringArrayField>

</my:ArrayFieldValue>

<my:MapFieldValue>

  <my:map1>
    <s:entry><s:key>1</s:key><s:value>hello</s:value></s:entry>
    <s:entry><s:key>2</s:key><s:value>world</s:value></s:entry>
  </my:map1>

  <my:map2>
    <s:e><s:k>1</s:k><s:v>java.lang.Integer</s:v></s:e>
    <s:e><s:k>2</s:k><s:v>java.lang.Long</s:v></s:e>
  </my:map2>

</my:MapFieldValue>
```

Type conversion is done automatically for all primitives and primitive wrappers, `Date`, `Calendar`,`Enum` and `Class` fields.

The use of EL to set field values is also supported:

```
<m:Report>
  <m:name>#{reportName}</m:name>
  <m:parameters>
    <s:key>#{paramName}</s:key>
    <s:value>#{paramValue}</s:key>
  </m:parameters>
</m:Report>
```

Internally, field values are set by wrapping the `InjectionTarget` for a bean. This means that the expressions are evaluated once, at bean creation time.

18.4.2. Inline Bean Declarations

Inline beans allow you to set field values to another bean that is declared inline inside the field declaration. This allows for the configuration of complex types with nestled classes. Inline beans can be declared inside both `<s:value>` and `<s:key>` elements, and may be used in both collections and simple field values. Inline beans must not have any qualifier annotations declared on the bean; instead Solder Config assigns them an artificial qualifier. Inline beans may have any scope, however the default `Dependent` scope is recommended.

```
<my:Knight>
 <my:sword>
   <value>
     <my:Sword type="sharp"/>
   </value>
 </my:sword>
 <my:horse>
   <value>
     <my:Horse>
       <my:name>
         <value>billy</value>
       </my:name>
       <my:shoe>
         <Inject/>
       </my:shoe>
     </my:Horse>
   </value>
 </my:horse>
```

</my:Knight>

18.5. Configuring methods

It is also possible to configure methods in a similar way to configuring fields:

```java
class MethodBean {

  public int doStuff() {
    return 1;
  }

  public int doStuff(MethodValueBean bean) {
    return bean.value + 1;
  }

  public void doStuff(MethodValueBean[][] beans) {
    /*do stuff */
  }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:java:ee"
    xmlns:my="urn:java:org.jboss.solder.config.xml.test.method">
  <my:MethodBean>

    <my:doStuff>
      <s:Produces/>
    </my:doStuff>

    <my:doStuff>
      <s:Produces/>
      <my:Qualifier1/>
      <s:parameters>
        <my:MethodValueBean>
          <my:Qualifier2/>
        </my:MethodValueBean>
      </s:parameters>
```

```
    </my:doStuff>

    <my:doStuff>
      <s:Produces/>
      <my:Qualifier1/>
      <s:parameters>
        <s:array dimensions="2">
          <my:Qualifier2/>
          <my:MethodValueBean/>
        </s:array>
      </s:parameters>
    </my:doStuff>


  </my:MethodBean>
</beans>
```

In this example, `MethodBean` has three methods. They are all named `doStuff`.

The first `<test:doStuff>` entry in the XML file configures the method that takes no arguments. The `<s:Produces>` element makes it into a producer method.

The next entry in the file configures the method that takes a `MethodValueBean` as a parameter and the final entry configures a method that takes a two dimensional array of `MethodValueBean`s as a parameter. For both of these methods, a qualifier was added to the method parameter and they were made into producer methods.

Method parameters are specified inside the `<s:parameters>` element. If these parameters have annotation children they are taken to be annotations on the parameter.

The corresponding Java declaration for the XML above would be:

```
class MethodBean {

 @Produces
 public int doStuff() {/*method body */}

 @Produces
 @Qualifier1
 public int doStuff(@Qualifier2 MethodValueBean param) {/*method body */}

 @Produces
 @Qualifier1
 public int doStuff(@Qualifier2 MethodValueBean[][] param) {/*method body */}
}
```

Array parameters can be represented using the `<s:array>` element, with a child element to represent the type of the array. E.g. `int method(MethodValueBean[] param);` could be configured via xml using the following:

```
<my:method>
  <s:array>
    <my:MethodValueBean/>
  </s:array>
</my:method>
```

> **Note**
>
> If a class has a field and a method of the same name then by default the field will be resolved. The exception is if the element has a child `<parameters>` element, in which case it is resolved as a method.

18.6. Configuring the bean constructor

It is also possible to configure the bean constructor in a similar manner. This is done with a `<s:parameters>` element directly on the bean element. The constructor is resolved in the same way methods are resolved. This constructor will automatically have the `@Inject` annotation applied to it. Annotations can be applied to the constructor parameters in the same manner as method parameters.

```
<my:MyBean>
  <s:parameters>
      <s:Integer>
        <my:MyQualifier/>
      </s:Integer>
  </s:parameters>
</my:MyBean>
```

The example above is equivalent to the following java:

```
class MyBean {
  @Inject
  MyBean(@MyQualifier Integer count)
  {
    ...
```

```
  }
}
```

18.7. Overriding the type of an injection point

It is possible to limit which bean types are available to inject into a given injection point:

```
class SomeBean
{
 public Object someField;
}
```

```xml
<my:SomeBean>
    <my:someField>
     <s:Inject/>
     <s:Exact>com.mydomain.InjectedBean</s:Exact>
    </my:someField>
</my:SomeBean>
```

In the example above, only beans that are assignable to InjectedBean will be eligible for injection into the field. This also works for parameter injection points. This functionality is part of Solder, and the @Exact annotation can be used directly in java.

18.8. Configuring Meta Annotations

It is possible to make existing annotations into qualifiers, stereotypes or interceptor bindings.

This configures a stereotype annotation SomeStereotype that has a single interceptor binding and is named:

```xml
<my:SomeStereotype>
 <s:Stereotype/>
 <my:InterceptorBinding/>
 <s:Named/>
</my:SomeStereotype>
```

This configures a qualifier annotation:

```xml
<my:SomeQualifier>
```

```
  <s:Qualifier/>
</my:SomeQualifier>
```

This configures an interceptor binding:

```
<my:SomeInterceptorBinding>
  <s:InterceptorBinding/>
</my:SomeInterceptorBinding>
```

18.9. Virtual Producer Fields

Solder XML Config supports configuration of virtual producer fields. These allow for configuration of resource producer fields, Solder generic bean and constant values directly via XML. For example:

```
<s:EntityManager>
  <s:Produces/>
  <s:PersistenceContext unitName="customerPu" />
</s:EntityManager>

<s:String>
  <s:Produces/>
  <my:VersionQualifier />
  <value>Version 1.23</value>
</s:String>
```

The first example configures a resource producer field. The second configures a bean of type String, with the qualifier `@VersionQualifier` and the value `'Version 1.23'`. The corresponding java for the above XML is:

```
class SomeClass
{

  @Produces
  @PersistenceContext(unitName="customerPu")
  EntityManager field1;

  @Produces
  @VersionQualifier
  String field2 = "Version 1.23";
```

```
}
```

Although these look superficially like normal bean declarations, the `<Produces>` declaration means it is treated as a producer field instead of a normal bean.

18.10. More Information

For further information, look at the units tests in the Solder XML Config distribution. Also see the XML-based metadata chapter in the *JSR-299 Public Review Draft* [http://jcp.org/aboutJava/ communityprocess/pr/jsr299/index.html], which is where this feature was originally proposed.

Introduction

The goal of Solder's Servlet integration features 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 Solder's exception handling chain and binding the `BeanManager` to a Servlet context attribute for convenient access.

Installation

19.1. 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:

```xml
<listener>
   <listener-class>org.jboss.solder.servlet.event.ServletEventBridgeListener</listener-class>
</listener>

<servlet>
   <servlet-name>Servlet Event Bridge Servlet</servlet-name>
   <servlet-class>org.jboss.solder.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>

<servlet-mapping>
   <servlet-name>Servlet Event Bridge Servlet</servlet-name>
   <url-pattern>/*</url-pattern>
</servlet-mapping>

<filter>
   <filter-name>Exception Filter</filter-name>
   <filter-class>org.jboss.solder.servlet.exception.CatchExceptionFilter</filter-class>
</filter>

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

<filter>
   <filter-name>Servlet Event Bridge Filter</filter-name>
   <filter-class>org.jboss.solder.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 Servlet event bridge to properly fire the `ServletContext` initialized event, the CDI runtime must be started at the time the 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 Servlet listener in `web.xml`.

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

Servlet event propagation

By including the Solder 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. Solder also fires additional lifecycle events not offered by the Servlet API, such as when the response is initialized and destroyed.

20.1. 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.solder.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 qualifiers:

```
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.

20.2. Application initialization

The servlet context initialized event described in the previous section provides an ideal opportunity to perform startup logic *as an alternative 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.solder.servlet.WebApplication` rather than the `ServletContext`. `WebApplication` is a informational object provided by Solder 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");
}
```

20.3. 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.solder.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	An HTTP servlet request is initialized or destroyed
@Initialized	javax.servlet.http.HttpServletRequest	An HTTP servlet request is initialized
@Destroyed	javax.servlet.http.HttpServletRequest	An 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 qualifier:

```
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.

20.4. Servlet response lifecycle events

The Servlet API does not provide a listener for accessing the lifecycle of a response. Therefore, Solder 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.solder.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	An HTTP servlet response is initialized or destroyed
@Initialized	javax.servlet.http.HttpServletResponse	An HTTP servlet response is initialized
@Destroyed	javax.servlet.http.HttpServletResponse	An HTTP 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 qualifier

```
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.

20.5. 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 Solder 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.solder.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!

20.6. 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.solder.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 qualifier

```
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.

20.7. 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.solder.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 one particular event, use a qualifier:

```
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

Solder 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 Solder exposes and how to inject them.

21.1. @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();
> ```

21.2. @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();
> ```

21.3. @Inject ServletContext

The `ServletContext` is made available as an application-scoped bean. It can be injected safely 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.

21.4. @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 safely 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.

21.5. @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 safely 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.

21.6. @Inject HttpSession

The `HttpSession` is made available as a request-scoped bean. It can be injected safely 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 Solder provides.

21.7. @Inject HttpSessionStatus

The `HttpSessionStatus` is a request-scoped bean that provides access to the status of the `HttpSession`. It can be injected safely 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();
}
```

21.8. @Inject @ContextPath

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

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

You can safely 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`.

21.9. @Inject List<Cookie>

The list of `Cookie` objects is made available as a request-scoped bean. It can be injected safely 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` instances by calling `getCookie()`.

21.10. @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();
> ```

21.11. @Inject @ServerInfo

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

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

The context path is retrieved from the `ServletContext`.

21.12. @Inject @Principal

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

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

Servlet Exception Handling Integration

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

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

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

22.2. Defining a exception handler for a web request

You can define an exception handler for a web request using the normal syntax of a Solder 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 Solder!");
    }
}
```

> **Note**
>
> `@WebRequest` may be added to limit handlers to only catch exceptions initiated by the Servlet integration.

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());
}
```

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, Solder 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
```

Solder also includes a provider that retrieves the `BeanManager` from this location. Anytime the Solder 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:

```java
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, among other available strategies. Refer to *Chapter 10, Obtaining a reference to the
BeanManager* for information on how to leverage the servlet context attribute provider to access
the `BeanManager` from outside the CDI environment.

Exception Handling - Introduction

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

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

24.1. How Solder's Exception Handling Works

Exception handling in Solder is based around the CDI eventing model. While the implementation of exception handlers may not be the same as a CDI event, and the programming model is not exactly the same as specifying a CDI event / observer, the concepts are very similar. Solder makes use of events for many of its features. Eventing is actually the only way to start using Solder's exception handling.

This event is fired either by the application or a Solder exception handling integration. Solder 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 the exception handling infrastructure.

The exception handling process remains mostly transparent to the developer. In most 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.

There are other events that are fired during the exception handling process that will allow great customization of the exception, stack trace, and status. This allows the application developer to have the most control possible while still following a defined workflow. These events and other advanced usages will be covered in the next chapter.

Exception Handling - Usage

25.1. Eventing into the exception handling framework

The entire exception handling process starts with an event. This helps keep your application minimally coupled to Solder, but also allows for further extension. Exception handling in Solder is all about letting you take care of exceptions the way that makes the most sense for your application. Events provide this delicate balance.

There are three means of firing the event to start the exception handling process:

- manual firing of the event

- using an interceptor

- module integration - no code needs to be changed

25.1.1. Manual firing of the event

Manually firing an event to use Solder's exception handling is primarily used in your own try/catch blocks. It's very painless and also easy. Let's examine an sample that might exist inside of a simple business logic lookup into an inventory database:

```
@Stateless
public class InventoryActions {
  @PersistenceContext private EntityManager em;

  @Inject private Event<ExceptionToCatch> catchEvent;                    ①

  public Integer queryForItem(Item item) {
   try {
     Query q = em.createQuery("SELECT i from Item i where i.id = :id");
     q.setParameter("id", item.getId());
     return q.getSingleResult();
   } catch (PersistenceException e) {
     catchEvent.fire(new ExceptionToCatch(e));                          ②
   }
  }
}
```

① The `Event` of generic type `ExceptionToCatch` is injected into your class for use later within a try/catch block.

(2)   The event is fired with a new instance of `ExceptionToCatch` constructed with the exception to be handled.

25.1.2. Using the `@ExceptionHandled` Interceptor

A CDI Interceptor has been added to help with integration of Solder exception handling into your application. It's used just like any interceptor, and must be enabled in the `beans.xml` file for your bean archive. This interceptor can be used at the class or method level.

This interceptor is a typical `AroundInvoke` interceptor and is invoked before the method (which in this case merely wraps the call to the intercepted method in a try / catch block). The intercepted method is called then, if an exception (actually a `Throwable`) occurs during execution of the intercepted method the exception is passed to Solder (without any qualifiers). Normal flow continues from there, however, take not of the following warning:

> ⚠️ **Warning**
>
> Using the interceptor may cause unexpected behavior to methods that call intercepted methods in which an exception occurs, please see the *API docs* [http://docs.jboss.org/seam/3/solder/latest/api/org/jboss/solder/exception/control/ExceptionHandledInterceptor.html] for more information about returns if an exception occurs.

25.2. Exception handlers

As an application developer (i.e., an end user of Solder's exception handling), 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.

> ℹ️ **Note**
>
> If there are no exception handlers for an exception, the exception is rethrown.

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 and subsequent chapters, you'll learn how to define an exception handler, explore how and when it gets invoked, modify an exception and a stack trace, and even extend exception handling

further through events that are fired during the handling workflow. We'll begin by covering the two annotations that are used to declare an exception handler, `@HandlesExceptions` and `@Handles`.

25.3. Exception handler annotations

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

25.3.1. @HandlesExceptions

The `@HandlesException` annotation is simply a marker annotation that instructs the Solder exception handling 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 at a later date.

25.3.2. @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`. Solder will discover all such methods at deployment time.

Let's look at an example. The following method is invoked for every exception that Solder processes and prints the exception message to stdout. (`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.markHandled();                                       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 (though it is not required to be the first parameter). 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 chain, as returned by `getException()`.

④ This handler does not modify the invocation of subsequent handlers, as designated by invoking `markHandled()` on `CaughtException`. As this is the default behavior, this line could be omitted.

The `@Handles` annotation must be placed on a 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 principles and guidelines as observers (such as invocation, injection of parameters, qualifiers, etc) with the following exceptions:

- a 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 precedence are invoked before handlers with lower precedence that handle the same exception type. The default precedence (if not specified) is 0.

- the type of the traversal mode (i.e., phase) during which the handler is invoked. The default traversal mode (if not specified) is `TraversalMode.DEPTH_FIRST`.

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 = TraversalMode.BREADTH_FIRST)      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 breadth first traversal mode. For more information on traversal, see the section *Section 25.5.1, "Traversal of exception type hierarchy"*.

③ This handler is qualified with `@WebRequest`. When Solder calculates the handler chain, it filters handlers based on the exception type and qualifiers. This handler will only be invoked for exceptions passed to Solder 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 Solder. 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 re-enables itself by invoking the `unmute()` method on the `CaughtException` instance.

Handlers must not throw checked exceptions, and should avoid throwing unchecked exceptions. Should a handler throw an unchecked exception it will propagate up the stack and all handling done via Solder will cease. Any exception that was being handled will be lost.

25.4. Exception chain 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 chain.

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

The first exception handlers to be invoked by Solder 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 Solder's exception handling a worthwhile tool.*

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

Consider a exception chain containing the following nested causes (from outer cause to root cause):

- EJBException

- PersistenceException

- SQLGrammarException

Solder 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`?

25.5. Exception handler ordering

While processing one of the causes in the exception chain, Solder 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.

25.5.1. Traversal of exception type hierarchy

Solder 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 in breadth first traversal mode, then gradually works down the type hierarchy toward handlers for the actual exception type, still in breadth first traversal. Once all breadth first traversal handlers have been

invoked, the process is reversed for depth first traversal, meaning the most specific handlers are notified first and Solder continues walking up the hierarchy tree.

There are two modes of this traversal:

- BREADTH_FIRST

- DEPTH_FIRST

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

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

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

1. `Throwable` (BREADTH_FIRST)

2. `Exception` (BREADTH_FIRST)

3. `IOException` (BREADTH_FIRST)

4. `SocketException` (BREADTH_FIRST)

5. `SocketException` (DEPTH_FIRST)

6. `IOException` (DEPTH_FIRST)

7. `Exception` (DEPTH_FIRST)

8. `Throwable` (DEPTH_FIRST)

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

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

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

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

25.5.2. Handler precedence

When Solder finds more than one handler for the same exception type, it orders the handlers by precedence. Handlers with higher precedence are executed before handlers with a lower precedence. If Solder 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 precedence:

```java
void handleIOExceptionFirst(@Handles(precedence = 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 precedence (100) than the second method, which has the default precedence (0).

To make specifying precedence values more convenient, Solder provides several built-in constants, available on the `Precedence` class:

- BUILT_IN = -100

- FRAMEWORK = -50

- DEFAULT = 0

- LOW = 50

- HIGH = 100

To summarize, here's how Solder 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 BREADTH_FIRST traversal

4. If multiple handlers for same type, invoke handlers with higher precedence first

5. Find handler for most specific handler marked for DEPTH_FIRST traversal

6. If multiple handlers for same type, invoke handlers with higher precedence first

7. Continue above steps for each exception in stack

25.6. APIs for exception information and flow control

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

- `CaughtException`

- `ExceptionStack`

25.6.1. 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 Solder is rethrown. Does not mark the exception as handled.

- `handled()` - marks the exception as handled and terminates further handling.

- `markHandled()` - default. Marks the exception as handled and proceeds with the rest of the handlers.

- `dropCause()` - 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 chain, unless it's explicitly marked as unmuted via the `unmute()` method on `CaughtException`.

25.6.2. ExceptionStack

`ExceptionStack` contains information about the exception causes relative to the current exception cause. It is also the source of the exception types the invoked handlers are

matched against. It is accessed in handlers by calling the method `getExceptionStack()` on the `CaughtException` object. Please see *API docs* [http://docs.jboss.org/seam/3/solder/latest/api/org/jboss/solder/exception/control/ExceptionStack.html] for more information, all methods are fairly self-explanatory.

> **Tip**
>
> This object is mutable and can be modified before any handlers are invoked by an observer:
>
> ```
> public void modifyStack(@Observes ExceptionStack stack) {
>  ...
> }
> ```
>
> Modifying the `ExceptionStack` may be useful to remove exception types that are effectively meaningless such as `EJBException`, changing the exception type to something more meaningful such as cases like `SQLException`, or wrapping exceptions as custom application exception types.

Exception handling - Advanced Features

26.1. Exception Modification

26.1.1. Introduction

At times it may be useful to change the exception to something a little more specific or meaningful before it is sent to handlers. Solder provides the means to make this happen. A prime use case for this behavior is a persistence-related exception coming from the database. Many times what we get from the database is an error number inside of a `SQLException`, which isn't very helpful.

26.1.2. Usage

Before any handlers are notified of an exception, Solder will raise an event of type `ExceptionStack`. This type contains all the exceptions in the chain, and will allow you to change the exception elements that will be used to notify handlers using the `setCauseElements(Collection)` method. Do not use any of the other methods as they only return copies of the chain.

> **Tip**
>
> When changing the exception, it is strongly recommended you keep the same stack trace for the exceptions you are changing. If the stack trace is not set then the new exception will not contain any stack information save from the time it was created, which is likely to be of little use to any handler.

26.2. Filtering Stack Traces

26.2.1. Introduction

Stack traces are an everyday occurrence for the Java developer, unfortunately the base stack trace isn't very helpful and can be difficult to understand and see the root problem. Solder helps make this easier by:

- turning the stack upside down and showing the root cause first, and

- allowing the stack trace to be filtered

The great part about all of this: it's done without a need for CDI! You can use it in a basic Java project, just include the Solder jar. There are four classes to be aware of when using filtering

- ExceptionStackOutput

- StackFrameFilter

- StackFrameFilterResult

- StackFrame

26.2.2. ExceptionStackOutput

There's not much to this, pass it the exception to print and the filter to use in the constructor and call `printTrace()` which returns a string -- the stack trace (filtered or not). If no filter is passed to the constructor, calling `printTrace()` will still unwrap the stack and print the root cause first. This can be used in place of `Throwable#printStackTrace()`, provided the returned string is actually printed to standard out or standard error.

26.2.3. StackFrameFilter

This is the workhorse interface that will need to be implemented to do any filtering for a stack trace. It only has one method: `public StackFrameFilterResult process(StackFrame frame)`. Further below are methods on `StackFrame` and `StackFrameFilterResult`. Some examples are included below to get an idea what can be done and how to do it.

26.2.4. StackFrameFilterResult

This is a simple enumeration of valid return values for the `process` method. Please see the *API docs* [http://docs.jboss.org/seam/3/solder/latest/api/org/jboss/solder/exception/filter/StackFrameFilterResult.html] for definitions of each value.

26.2.5. StackFrame

This contains methods to help aid in determining what to do in the filter, it also allows you to completely replace the `StackTraceElement` if desired. The four "mark" methods deal with marking a stack trace and are used if "folding" a stack trace is desired, instead of dropping the frame. The `StackFrame` will allow for multiple marks to be set. The last method, `getIndex()`, will return the index of the `StackTraceElement` from the exception.

Example 26.1. Terminate

```
@Override
    public StackFrameFilterResult process(StackFrame frame) {
      return StackFrameFilterResult.TERMINATE;
    }
```

This example will simply show the exception, no stack trace.

Example 26.2. Terminate After

```
@Override
    public StackFrameFilterResult process(StackFrame frame) {
        return StackFrameFilterResult.TERMINATE_AFTER;
    }
```

This is similar to the previous example, save the first line of the stack is shown.

Example 26.3. Drop Remaining

```
@Override
    public StackFrameFilterResult process(StackFrame frame) {
        if (frame.getIndex() >= 5) {
            return StackFrameFilterResult.DROP_REMAINING;
        }
        return StackFrameFilterResult.INCLUDE;
    }
```

This filter drops all stack elements after the fifth element.

Example 26.4. Folding

```
@Override
    public StackFrameFilterResult process(StackFrame frame) {
        if (frame.isMarkSet("reflections.invoke")) {
            if (frame.getStackTraceElement().getClassName().contains("java.lang.reflect")) {
                frame.clearMark("reflections.invoke");
                return StackFrameFilterResult.INCLUDE;
            }
            else if (frame.getStackTraceElement().getMethodName().startsWith("invoke")) {
                return StackFrameFilterResult.DROP;
            }
        }

        if (frame.getStackTraceElement().getMethodName().startsWith("invoke")) {
            frame.mark("reflections.invoke");
            return StackFrameFilterResult.DROP;
        }

        return StackFrameFilterResult.INCLUDE;
```

```
    }
```

Certainly the most complicated example, however, this shows a possible way of "folding" a stack trace. In the example any internal reflection invocation methods are folded into a single `java.lang.reflect.Method.invoke()` call, no more internal com.sun calls in the trace.

Exception Handling - Framework Integration

Integration of Solder's exception handling 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

27.1. 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 Solder's exception handling system and allow handlers to be invoked. This is the typical way of using Solder to handle exceptions. Of course, it doesn't stop the application developer from firing their own `ExceptionToCatch` within a catch block.

> **Tip**
>
> The integration should check to see if the exception was handled and rethrow the exception if it was not handled. It should also wrap the firing of the event in a try catch, and unwrap any exceptions that are thrown. This exception should be `javax.enterprise.event.ObserverException` and should wrap the exception that should be rethrown.

27.2. Default Handlers and Qualifiers

27.2.1. Default Handlers

An integration with Solder 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.

27.2.2. Qualifiers

Solder supports qualifiers for the `CaughtException`. To add qualifiers to be used when notifying handlers, the qualifiers must be added to the `ExceptionToCatch` instance 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.

27.3. Supporting ServiceHandlers

ServiceHandlers make for a very easy and concise way to define exception handlers. The following example is a possible usage of ServiceHandlers within a JAX-RS application:

```
@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 Solder integration, however, there may be situations where `ServiceHandler`s 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()`).

27.4. Programmatic Handler Registration

Handlers can be registered programatically at runtime instead of solely at deploy time. This done very simply by injecting `HandlerMethodContainer` and calling `registerHandlerMethod(HandlerMethod)`.

`HandlerMethod` has been relaxed in this version as well, and is not tied directly to Java. It is therefore feasible handlers written using other JVM based languages could be easily registered and participate in exception handling.

Exception Handling - Glossary

E

Exception Chain

An exception chain is made up of many different exceptions or causes until the root exception is found at the bottom of the chain. When all of the causes are removed or looked at this forms the causing container. The container may be traversed either ascending (root cause first) or descending (outer most first).

H

Handler Bean

A CDI enabled Bean which contains handler methods. Annotated with the `@HandlesExceptions` annotation.
See Also *Handler Method*.

Handler Method

A method within a handler bean which is marked as a handler using the `@Handler`s on an argument, which must be an instance of `CaughtException`. Handler methods typically are public with a void return. Other parameters of the method will be treated as injection points and will be resolved via CDI and injected upon invocation.
See Also *Handler Bean*.

Part II. Seam Persistence

Seam Persistence Reference

Seam provides extensive support for the two most popular persistence architectures for Java: Hibernate3, and the Java Persistence API introduced with EJB 3.0. Seam's unique state-management architecture allows the most sophisticated ORM integration of any web application framework.

28.1. Introduction

Seam grew out of the frustration of the Hibernate team with the statelessness typical of the previous generation of Java application architectures. The state management architecture of Seam was originally designed to solve problems relating to persistence — in particular problems associated with *optimistic transaction processing*. Scalable online applications always use optimistic transactions. An atomic (database/JTA) level transaction should not span a user interaction unless the application is designed to support only a very small number of concurrent clients. But almost all interesting work involves first displaying data to a user, and then, slightly later, updating the same data. So Hibernate was designed to support the idea of a persistence context which spanned an optimistic transaction.

Unfortunately, the so-called "stateless" architectures that preceded Seam and EJB 3.0 had no construct for representing an optimistic transaction. So, instead, these architectures provided persistence contexts scoped to the atomic transaction. Of course, this resulted in many problems for users, and is the cause of the number one user complaint about Hibernate: the dreaded `LazyInitializationException`. What we need is a construct for representing an optimistic transaction in the application tier.

EJB 3.0 recognizes this problem, and introduces the idea of a stateful component (a stateful session bean) with an *extended persistence context* scoped to the lifetime of the component. This is a partial solution to the problem (and is a useful construct in and of itself) however there are two problems:

- The lifecycle of the stateful session bean must be managed manually via code in the web tier (it turns out that this is a subtle problem and much more difficult in practice than it sounds).

- Propagation of the persistence context between stateful components in the same optimistic transaction is possible, but tricky.

Seam solves the first problem by providing conversations, and stateful session bean components scoped to the conversation. (Most conversations actually represent optimistic transactions in the data layer.) This is sufficient for many simple applications (such as the Seam booking demo) where persistence context propagation is not needed. For more complex applications, with many loosely-interacting components in each conversation, propagation of the persistence context across components becomes an important issue. So Seam extends the persistence context management model of EJB 3.0, to provide conversation-scoped extended persistence contexts.

28.2. Getting Started

To get started with Seam persistence you need to add the `seam-persistence.jar` and the `seam-solder.jar` to you deployment. If you are in a java SE environment you will probably also require `seam-xml.jar` as well for configuration purposes. The relevant maven configuration is as follows:

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

<dependency>
  <groupId>org.jboss.seam.persistence</groupId>
  <artifactId>seam-persistence-impl</artifactId>
  <version>${seam.persistence.version}</version>
</dependency>

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

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

You will also need to have a JPA provider on the classpath. If you are using java EE this is taken care of for you. If not, we recommend hibernate.

```xml
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>3.5.1-Final</version>
</dependency>
```

28.3. Transaction Management

Unlike EJB session beans CDI beans are not transactional by default. Seam brings declarative transaction management to CDI beans by enabling them to use `@TransactionAttribute`. Seam also provides the `@Transactional` annotation, for environments where java EE APIs are not present.

28.3.1. Configuration

In order to enable declarative transaction management for managed beans you need to list the transaction interceptor in beans.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://docs.jboss.org/cdi/beans_1_0.xsd">
  <interceptors>
    <class>org.jboss.seam.transaction.TransactionInterceptor</class>
  </interceptors>
</beans>
```

If you are in a Java EE 6 environment then you are good to go, no additional configuration is required.

If you are not in an EE environment you may need to configure some things with seam-xml. You may need the following entries in your beans.xml file:

```xml
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:t="urn:java:org.jboss.seam.transaction"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/javaee
    http://docs.jboss.org/cdi/beans_1_0.xsd">

  <t:SeSynchronizations>
    <s:modifies/>
  </t:SeSynchronizations>

  <t:EntityTransaction>
    <s:modifies />
```

```
    </t:EntityTransaction>

</beans>
```

Let's look at these individually.

```
<t:SeSynchronizations>
  <s:modifies/>
</t:SeSynchronizations>
```

Seam will attempt to use JTA synchronizations if possible. If not then you need to install the `SeSynchronzations` bean to allow seam to handle synchronizations manually. Synchronizations allow seam to respond to transaction events such as `beforeCompletion()` and`afterCompletion()`, and are needed for the proper operation of the *Seam Managed Persistence Context*.

```
<t:EntityTransaction>
  <s:modifies />
</t:EntityTransaction>
```

By default seam will attempt to look up `java:comp/UserTransaction` from JNDI (or alternatively retrieve it from the `EJBContext` if a container managed transaction is active). Installing `EntityTransaction` tells seam to use the JPA `EntityTransaction` instead. To use this you must have a *Seam Managed Persistence Context* installed with qualifier`@Default`.

If your entity manager is installed with a different qualifier, then you need to use the following configuration (this assumes that `my` has been bound to the namespace that contains the appropriate qualifier, see the Seam Config XML documentation for more details):

```
<t:EntityTransaction>
  <s:modifies />
  <t:entityManager>
    <my:SomeQualifier/>
  </tentityManager>
</t:EntityTransaction>
```

> **Note**
>
> You should avoid `EntityTransaction` if you have more than one persistence unit in your application. Seam does not support installing multiple `EntityTransaction` beans, and the `EntityTransaction` interface does not support two phase commit, so unless you are careful you may have data consistency issues. If you need multiple persistence units in your application then we highly recommend using an EE 6 compatible server, such as JBoss 6.

28.3.2. Declarative Transaction Management

Seam adds declarative transaction support to managed beans. Seam re-uses the EJB `@TransactionAttribute` for this purpose, however it also provides an alternative `@Transactional` annotation for environments where the EJB API's are not available. An alternative to `@ApplicationException`, `@SeamApplicationException` is also provided. Unlike EJBs, managed beans are not transactional by default, you can change this by adding the `@TransactionAttribute` to the bean class.

Unlike in Seam 2, transactions will not roll back whenever a non-application exception propagates out of a bean, unless the bean has the transaction interceptor enabled.

If you are using seam managed transactions as part of the seam-faces module you do not need to worry about declarative transaction management. Seam will automatically start a transaction for you at the start of the faces request, and commit it before the render response phase.

> **Warning**
>
> `@SeamApplicationException` will not control transaction rollback when using EJB container managed transactions. If you are in an EE environment then you should always use the EJB API's, namely `@TransactionAttribute` and `@ApplicationException`.

> **Note**
>
> `TransactionAttributeType.REQUIRES_NEW` and `TransactionAttributeType.NOT_SUPPORTED` are not yet supported on managed beans.

Let's have a look at some code. Annotations applied at a method level override annotations applied at the class level.

```
@TransactionAttribute /*Defaults to TransactionAttributeType.REQUIRED */
```

```java
class TransactionaBean
{

/* This is a transactional method, when this method is called a transaction
 * will be started if one does not already exist.
 * This behavior is inherited from the @TransactionAttribute annotation on
 * the class.
 */
void doWork()
{
...
}


/* A transaction will not be started for this method, however it */
/* will not complain if there is an existing transaction active. */
@TransactionAttributeType(TransactionAttributeType.SUPPORTED)
void doMoreWork()
{
...
}

/* This method will throw an exception if there is no transaction active when */
/* it is invoked. */

@TransactionAttributeType(TransactionAttributeType.MANDATORY)
void doEvenMoreWork()
{
...
}

/* This method will throw an exception if there is a transaction active when */
/* it is invoked. */
@TransactionAttributeType(TransactionAttributeType.NOT_SUPPORTED)
void doOtherWork()
{
...
}
}
```

28.4. Seam-managed persistence contexts

If you're using Seam outside of a Java EE environment, you can't rely upon the container to manage the persistence context lifecycle for you. Even if you are in an EE environment, you might have a complex application with many loosely coupled components that collaborate together in the scope of a single conversation, and in this case you might find that propagation of the persistence context between component is tricky and error-prone.

In either case, you'll need to use a *managed persistence context* (for JPA) or a *managed session* (for Hibernate) in your components. A Seam-managed persistence context is just a built-in Seam component that manages an instance of `EntityManager` or `Session` in the conversation (or any other) context. You can inject it with `@Inject`.

28.4.1. Using a Seam-managed persistence context with JPA

```
@ExtensionManaged
        @Produces
        @PersistenceUnit
        @ConversationScoped
        EntityManagerFactory producerField;
```

This is just an ordinary resource producer field as defined by the CDI specification, however the presence of the `@ExtensionManaged` annotation tells seam to create a seam managed persistence context from this `EntityManagerFactory`. This managed persistence context can be injected normally, and has the same scope and qualifiers that are specified on the resource producer field.

This will work even in a SE environment where `@PersistenceUnit` injection is not normally supported. This is because the seam persistence extensions will bootstrap the `EntityManagerFactory` for you.

Now we can have our `EntityManager` injected using:

```
@Inject EntityManager entityManager;
```

> **Note**
>
> The more eagle eyed among you may have noticed that the resource producer field appears to be conversation scoped, which the CDI specification does not require containers to support. This is actually not the case, as the @ConversationScoped

annotation is removed by the seam persistence portable extension. It only specifies the scope of the created SMPC, not the EntityManagerFactory.

> ⚠️ **Warning**
>
> If you are using EJB3 and mark your class or method `@TransactionAttribute(REQUIRES_NEW)` then the transaction and persistence context shouldn't be propagated to method calls on this object. However as the Seam-managed persistence context is propagated to any component within the conversation, it will be propagated to methods marked `REQUIRES_NEW`. Therefore, if you mark a method `REQUIRES_NEW` then you should access the entity manager using @PersistenceContext.

28.4.2. Seam-managed persistence contexts and atomic conversations

Persistence contexts scoped to the conversation allows you to program optimistic transactions that span multiple requests to the server without the need to use the `merge()` operation , without the need to re-load data at the beginning of each request, and without the need to wrestle with the `LazyInitializationException` or `NonUniqueObjectException`.

As with any optimistic transaction management, transaction isolation and consistency can be achieved via use of optimistic locking. Fortunately, both Hibernate and EJB 3.1 make it very easy to use optimistic locking, by providing the `@Version` annotation.

By default, the persistence context is flushed (synchronized with the database) at the end of each transaction. This is sometimes the desired behavior. But very often, we would prefer that all changes are held in memory and only written to the database when the conversation ends successfully. This allows for truly atomic conversations. Unfortunately there is currently no simple, usable and portable way to implement atomic conversations using EJB 3.1 persistence. However, Hibernate provides this feature as a vendor extension to the `FlushModeType`s defined by the specification, and it is our expectation that other vendors will soon provide a similar extension.

28.4.3. Using EL in EJB-QL/HQL

Seam proxies the `EntityManager` or `Session` object whenever you use a Seam-managed persistence context. This lets you use EL expressions in your query strings, safely and efficiently. For example, this:

```
User user = em.createQuery("from User where username=#{user.username}")
    .getSingleResult();
```

is equivalent to:

```
User user = em.createQuery("from User where username=:username")
  .setParameter("username", user.getUsername())
  .getSingleResult();
```

Of course, you should never, ever write it like this:

```
User user = em.createQuery("from User where username=" + user.getUsername()) //BAD!
  .getSingleResult();
```

(It is inefficient and vulnerable to SQL injection attacks.)

> ⚠️ **Warning**
>
> This only works with seam managed persistence contexts, not persistence contexts that are injected with `@PersistenceContext`.

28.4.4. Setting up the EntityManager

Sometimes you may want to perform some additional setup on the `EntityManager` after it has been created. For example, if you are using Hibernate you may want to set a filter. Seam persistence fires a `SeamManagedPersistenceContextCreated` event when a Seam managed persistence context is created. You can observe this event and perform any setup you require in an observer method. For example:

```
public void setupEntityManager(@Observes SeamManagedPersistenceContextCreated
        event) {
        Session session = (Session)event.getEntityManager().getDelegate();
        session.enableFilter("myfilter");
        }
```

Part III. Seam Transaction

Seam Transaction Reference

29.1. Introduction

Part IV. Seam Security

Security - Introduction

30.1. Overview

The Seam Security module provides a number of useful features for securing your Java EE application, which are briefly summarised in the following sections. The rest of the chapters contained in this documentation each focus on one major aspect of each of the following features.

30.1.1. Authentication

Authentication is the act of establishing, or confirming, the identity of a user. In many applications a user confirms their identity by providing a username and password (also known as their *credentials*). Seam Security allows the developer to control how users are authenticated, by providing a flexible *Authentication API* that can be easily configured to allow authentication against any number of sources, including but not limited to databases, LDAP directory servers or some other external authentication service.

If none of the built-in authentication providers are suitable for your application, then it is also possible to write your own custom Authenticator implementation.

30.1.2. Identity Management

Identity Management is a set of useful APIs for managing the users, groups and roles within your application. The identity management features in Seam are provided by PicketLink IDM, and allow you to manage users stored in a variety of backend security stores, such as in a database or LDAP directory.

30.1.3. External Authentication

Seam Security contains an external authentication sub-module that provides a number of features for authenticating your application users against external authentication services, such as OpenID and SAML.

30.1.4. Authorization

While *authentication* is used to confirm the identity of the user, *authorization* is used to control which actions a user may perform within your application. Authorization can be roughly divided into two categories; coarse-grained and fine-grained. An example of a coarse-grained restriction is allowing only members of a certain group or role to perform a privileged operation. A fine-grained restriction on the other hand may allow only a certain individual user to perform a specific action on a specific object within your application.

There are also rule-based permissions, which bridge the gap between fine-grained and coarse-grained restrictions. These permissions may be used to determine a user's privileges based on certain business logic.

30.2. Configuration

30.2.1. Maven Dependencies

The Maven artifacts for all Seam modules are hosted within the JBoss Maven repository. Please refer to the *Maven Getting Started Guide* [http://community.jboss.org/wiki/MavenGettingStarted-Users] for information about configuring your Maven installation to use the JBoss repository.

To use Seam Security within your Maven-based project, it is advised that you import the Seam BOM (Bill of Materials) which declares the versions for all Seam modules. First declare a property value for `${seam.version}` as follows:

```
<properties>
  <seam.version>3.1.0.Final</seam.version>
</properties>
```

You can check the *JBoss Maven Repository* [https://repository.jboss.org/nexus/content/groups/public/org/jboss/seam/seam-bom/] directly to determine the latest version of the Seam BOM to use.

Now add the following lines to the list of dependencies within the `dependencyManagement` section of your project's `pom.xml` file:

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>seam-bom</artifactId>
  <version>${seam.version}</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

Once that is done, add the following dependency (no version is required as it comes from `seam-bom`):

```
<dependency>
  <groupId>org.jboss.seam.security</groupId>
  <artifactId>seam-security</artifactId>
</dependency>
```

If you wish to use the external authentication module in your application to allow authentication using OpenID or SAML, then add the following dependency also:

```
<dependency>
 <groupId>org.jboss.seam.security</groupId>
 <artifactId>seam-security-external</artifactId>
</dependency>
```

30.2.2. Enabling the Security Interceptor

To enable many of the features of Seam Security, the Security interceptor must be configured in your application's `beans.xml` file. Add the following configuration to your `beans.xml` to enable the Security Interceptor:

```
<interceptors>
   <class>org.jboss.seam.security.SecurityInterceptor</class>
</interceptors>
```

Security - Authentication

31.1. Basic Concepts

The majority of the Security API is centered around the `Identity` bean. This bean represents the identity of the current user, the default implementation of which is a session-scoped, named bean. This means that once logged in, a user's identity is scoped to the lifecycle of their current session. The two most important methods that you need to know about at this stage in regard to authentication are `login()` and `logout()`, which as the names suggest are used to log the user in and out, respectively.

As the default implementation of the `Identity` bean is named, it may be referenced via an EL expression, or be used as the target of an EL action. Take the following JSF code snippet for example:

```
<h:commandButton action="#{identity.login}" value="Log in"/>
```

This JSF command button would typically be used in a login form (which would also contain inputs for the user's username and password) that allows the user to log into the application.

> **Note**
>
> The bean type of the `Identity` bean is `org.jboss.seam.security.Identity`. This interface is what you should inject if you need to access the `Identity` bean from your own beans. The default implementation is `org.jboss.seam.security.IdentityImpl`.

The other important bean to know about right now is the `Credentials` bean. Its' purpose is to hold the user's credentials (such as their username and password) before the user logs in. The default implementation of the `Credentials` bean is also a session-scoped, named bean (just like the `Identity` bean).

The `Credentials` bean has two properties, `username` and `credential` that are used to hold the current user's username and credential (e.g. a password) values. The default implementation of the `Credentials` bean provides an additional convenience property called `password`, which may be used in lieu of the `credential` property when a simple password is required.

> **Note**
>
> The bean type of the `Credential` bean is `org.jboss.seam.security.Credentials`. The default implementation for this

bean type is `org.jboss.seam.security.CredentialsImpl`. Also, as credentials may come in many forms (such as passwords, biometric data such as that from a fingerprint reader, etc) the `credential` property of the `Credentials` bean must be able to support each variation, not just passwords. To allow for this, any credential that implements the `org.picketlink.idm.api.Credential` interface is a valid value for the `credential` property.

31.2. Built-in Authenticators

The Seam Security module provides the following built-in `Authenticator` implementations:

- `org.jboss.seam.security.jaas.JaasAuthenticator` - used to authenticate against a JAAS configuration defined by the container.

- `org.jboss.seam.security.management.IdmAuthenticator` - used to authenticate against an Identity Store using the Identity Management API. See the Identity Management chapter for details on how to configure this authenticator.

- `org.jboss.seam.security.external.openid.OpenIdAuthenticator` (provided by the external module) - used to authenticate against an external OpenID provider, such as Google, Yahoo, etc. See the External Authentication chapter for details on how to configure this authenticator.

31.3. Which Authenticator will Seam use?

The `Identity` bean has an `authenticatorClass` property, which if set will be used to determine which `Authenticator` bean implementation to invoke during the authentication process. This property may be set by configuring it with a predefined authenticator type, for example by using Solder XML Config. The following XML configuration example shows how you would configure the `Identity` bean to use the `com.acme.MyCustomerAuthenticator` bean for authentication:

```xml
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee    http://jboss.org/schema/cdi/beans_1_0.xsd">

  <security:IdentityImpl>
    <s:modifies/>
                            <security:authenticatorClass>com.acme.MyCustomAuthenticator</security:authenticatorClass>
  </security:IdentityImpl>
```

```
</beans>
```

Alternatively, if you wish to be able to select the `Authenticator` to authenticate with by specifying the name of the `Authenticator` implementation (i.e. for those annotated with the `@Named` annotation), the `authenticatorName` property may be set instead. This might be useful if you wish to offer your users the choice of how they would like to authenticate, whether it be through a local user database, an external OpenID provider, or some other method.

The following example shows how you might configure the `authenticatorName` property with the Seam Config module:

```xml
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee    http://jboss.org/schema/cdi/
beans_1_0.xsd">
  <security:IdentityImpl>
    <s:modifies/>
    <security:authenticatorName>openIdAuthenticator</security:authenticatorName>
  </security:IdentityImpl>
</beans>
```

If neither the `authenticatorClass` or `authenticatorName` properties are set, then the authentication process with automatically use a custom `Authenticator` implementation, if the developer has provided one (and only one) within their application.

If neither property is set, and the user has not provided a custom `Authenticator`, then the authentication process will fall back to the Identity Management API to attempt to authenticate the user.

31.4. Writing a custom Authenticator

All `Authenticator` implementations must implement the `org.jboss.seam.security.Authenticator` interface. This interface defines the following methods:

```java
public interface Authenticator {
  void authenticate();
  void postAuthenticate();
  User getUser();
  AuthenticationStatus getStatus();
```

```
}
```

The `authenticate()` method is invoked during the authentication process and is responsible for performing the work necessary to validate whether the current user is who they claim to be.

The `postAuthenticate()` method is invoked after the authentication process has already completed, and may be used to perform any post-authentication business logic, such as setting session variables, logging, auditing, etc.

The `getUser()` method should return an instance of `org.picketlink.idm.api.User`, which is generally determined during the authentication process.

The `getStatus()` method must return the current status of authentication, represented by the `AuthenticationStatus` enum. Possible values are `SUCCESS`, `FAILURE` and `DEFERRED`. The `DEFERRED` value should be used for special circumstances, such as asynchronous authentication as a result of authenticating against a third party as is the case with OpenID, etc.

The easiest way to get started writing your own custom authenticator is to extend the `org.jboss.seam.security.BaseAuthenticator` abstract class. This class implements the `getUser()` and `getStatus()` methods for you, and provides `setUser()` and `setStatus()` methods for setting both the user and status values.

> ⚠️ **Warning**
>
> An `Authenticator` implementation cannot be a stateless session bean.

To access the user's credentials from within the `authenticate()` method, you can inject the `Credentials` bean like so:

```
@Inject Credentials credentials;
```

Once the credentials are injected, the `authenticate()` method is responsible for checking that the provided credentials are valid. Here is a complete example:

```java
public class SimpleAuthenticator extends BaseAuthenticator implements Authenticator {
  @Inject Credentials credentials;

  @Override
  public void authenticate() {
    if ("demo".equals(credentials.getUsername()) &&
        credentials.getCredential() instanceof PasswordCredential &&
        "demo".equals(((PasswordCredential) credentials.getCredential()).getValue())) {
      setStatus(AuthenticationStatus.SUCCESS);
```

```
      setUser(new SimpleUser("demo"));
   }
  }
}
```

> **Note**
>
> The above code was taken from the simple authentication example, included in
> the Seam Security distribution.

In the above code, the `authenticate()` method checks that the user has provided a username of
demo and a password of *demo*. If so, the authentication is deemed as successful and the status is
set to `AuthenticationStatus.SUCCESS`, and a new `SimpleUser` instance is created to represent
the authenticated user.

> **Warning**
>
> The `Authenticator` implementation *must* return a non-null value when `getUser()`
> is invoked if authentication is successful. Failure to return a non-null value will result
> in an `AuthenticationException` being thrown.

Security - Identity Management

32.1. Overview

Identity Management is a feature that allows you to manage the users, groups and roles in your application. The Identity Management features in Seam Security are provided by *PicketLink IDM* [http://www.jboss.org/picketlink/IDM]. The best place to find more information about PicketLink IDM is the reference documentation, available *here* [http://anonsvn.jboss.org/repos/picketlink/idm/downloads/docs/1.0.0.GA/ReferenceGuide/en-US/html_single/index.html].

PicketLink provides two identity store implementations to allow you to use Hibernate or LDAP to store identity-related data (please refer to the PicketLink IDM documentation for details on configuring these). Seam Security provides an additional implementation called `JpaIdentityStore`, which allows you to store your identity data using JPA.

In a Seam-based application it probably makes more sense to use the standards-based `JpaIdentityStore` rather than `HibernateIdentityStore`, as you will most likely be running in an Java EE container that supports JPA. `JpaIdentityStore` is an implementation of the PicketLink `IdentityStore` interface, provided by Seam Security. This identity store allows you to store your identity model inside a relational database, accessible via JPA. It provides an immense amount of flexibility in the way you define your identity model, and in most cases should be compatible with existing database schemas.

> **Note**
>
> See the idmconsole example application (included in the Seam distribution) for a demonstration of Seam's Identity Management features.

32.2. Configuring Seam to use Identity Management with JPA

Like all authentication providers in Seam, Identity Management is supported via a concrete `Authenticator` implementation called `IdmAuthenticator`. To use Identity Management in your own application, you don't need to do anything! Simply don't configure any authenticator, and as long as you have an identity store configured (see the next section), the Identity Management API will be used to authenticate automatically.

32.2.1. Recommended database schema

While `JpaIdentityStore` should be compatible with a large variety of database schemas, the following diagram displays the recommended database schema to use:
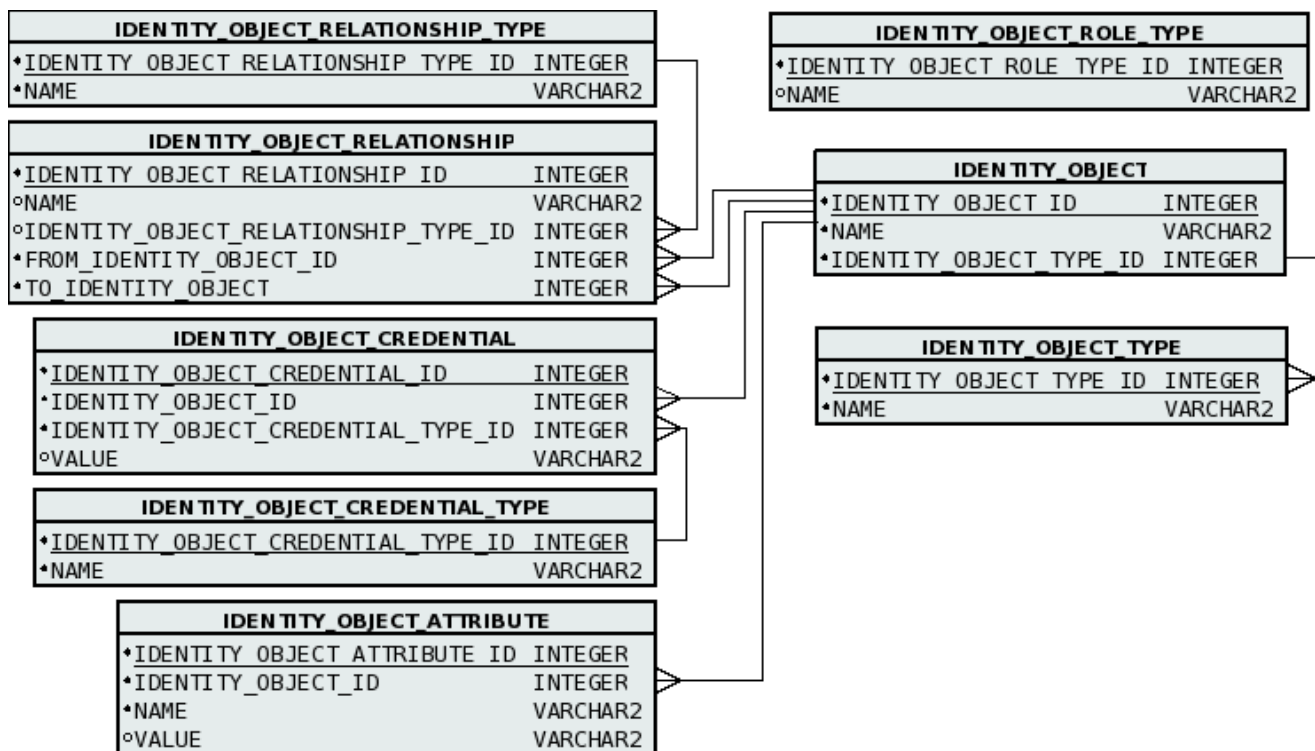
32.2.2. The `@IdentityEntity` and `@IdentityProperty` annotations

Seam Security provides two annotations for configuring your entity beans for use with `JpaIdentityStore`. The first, `@IdentityEntity` is a class annotation used to mark an entity bean so that `JpaIdentityStore` knows it contains identity-related data. It has a single member of type `EntityType`, that tells `JpaIdentityStore` what type of identity data it contains. Possible values are:

- IDENTITY_OBJECT

- IDENTITY_CREDENTIAL

- IDENTITY_RELATIONSHIP

- IDENTITY_ATTRIBUTE

- IDENTITY_ROLE_NAME

The second one, `IdentityProperty`, is a field or method annotation which is used to configure which properties of the bean contain identity values. This annotation declares two values, `value` and `attributeName`:

```
package org.jboss.seam.security.annotations.management;

public @interface IdentityProperty {
  PropertyType value();
```

```
  String attributeName() default "";
}
```

The `value()` member is of type `PropertyType`, which is an enum that defines the following values:

```
public enum PropertyType {
  NAME, TYPE, VALUE, RELATIONSHIP_FROM, RELATIONSHIP_TO, CREDENTIAL,
  CREDENTIAL_TYPE, ATTRIBUTE }
```

By placing the `IdentityProperty` annotation on various fields of your entity beans, `JpaIdentityStore` can determine how identity-related data must be stored within your database tables.

In the following sections we'll look at how each of the main entities are configured.

32.2.3. Identity Object

Let's start by looking at identity object. In the recommended database schema, the `IDENTITY_OBJECT` table is responsible for storing objects such as users and groups. This table may be represented by the following entity bean:

```
@Entity
@IdentityEntity(IDENTITY_OBJECT)
public class IdentityObject implements Serializable {
    @Id @GeneratedValue private Long id;

    @IdentityProperty(PropertyType.NAME)
    private String name;

    @ManyToOne @IdentityProperty(PropertyType.TYPE)
    @JoinColumn(name = "IDENTITY_OBJECT_TYPE_ID")
    private IdentityObjectType type;

    // snip getter and setter methods
}
```

In the above code both the `name` and `type` fields are annotated with `@IdentityProperty`. This tells `JpaIdentityStore` that these two fields are significant in terms of identity management-related state. By annotating the `name` field with `@IdentityProperty(PropertyType.NAME)`, `JpaIdentityStore` knows that this field is used to store the *name* of the identity object. Likewise, the `@IdentityProperty(PropertyType.TYPE)` annotation on the `type` field indicates that the value of this field is used to represent the *type* of identity object.

The `IdentityObjectType` entity is simply a lookup table containing the names of the valid identity types. The field representing the actual name of the type itself should be annotated with `@IdentityProperty(PropertyType.NAME)`:

```java
@Entity
public class IdentityObjectType implements Serializable {

  @Id @GeneratedValue private Long id;
  @IdentityProperty(PropertyType.NAME) private String name;

  // snip getter and setter methods
}
```

32.2.4. Credential

The credentials table is used to store user credentials, such as passwords. Here's an example of an entity bean configured to store identity object credentials:

```java
@Entity
@IdentityEntity(IDENTITY_CREDENTIAL)
public class IdentityObjectCredential implements Serializable {
  @Id @GeneratedValue private Long id;

  @ManyToOne @JoinColumn(name = "IDENTITY_OBJECT_ID")
  private IdentityObject identityObject;

  @ManyToOne @IdentityProperty(PropertyType.TYPE)
  @JoinColumn(name = "CREDENTIAL_TYPE_ID")
  private IdentityObjectCredentialType type;

  @IdentityProperty(PropertyType.VALUE)
  private String value;

  // snip getter and setter methods
}
```

The `IdentityObjectCredentialType` entity is used to store a list of valid credential types. Like `IdentityObjectType`, it is a simple lookup table with the field representing the credential type name annotated with `@IdentityProperty(PropertyType.NAME)`:

```java
@Entity
```

```
public class IdentityObjectCredentialType implements Serializable
{
  @Id @GeneratedValue private Long id;

  @IdentityProperty(PropertyType.NAME)
  private String name;

  // snip getter and setter methods
}
```

32.2.5. Identity Object Relationship

The relationship table stores associations between identity objects. Here's an example of an entity bean that has been configured to store identity object relationships:

```
@Entity
@IdentityEntity(IDENTITY_RELATIONSHIP)
public class IdentityObjectRelationship implements Serializable
{
  @Id @GeneratedValue private Long id;

  @IdentityProperty(PropertyType.NAME)
  private String name;

  @ManyToOne @IdentityProperty(PropertyType.TYPE) @JoinColumn(name = "RELATIONSHIP_TYPE_ID")
  private IdentityObjectRelationshipType relationshipType;

  @ManyToOne @IdentityProperty(PropertyType.RELATIONSHIP_FROM) @JoinColumn(name = "FROM_IDENTI
  private IdentityObject from;

  @ManyToOne @IdentityProperty(PropertyType.RELATIONSHIP_TO) @JoinColumn(name = "TO_IDENTITY_ID'
  private IdentityObject to;

  // snip getter and setter methods
}
```

The `name` property is annotated with `@IdentityProperty(PropertyType.NAME)` to indicate that this field contains the name value for named relationships. An example of a named relationship is a role, which uses the `name` property to store the role type name.

The `relationshipType` property is annotated with `@IdentityProperty(PropertyType.TYPE)` to indicate that this field represents the type of relationship. This is typically a value in a lookup table.

The `from` property is annotated with `@IdentityProperty(PropertyType.RELATIONSHIP_FROM)` to indicate that this field represents the `IdentityObject` on the *from* side of the relationship.

The `to` property is annotated with `@IdentityProperty(PropertyType.RELATIONSHIP_TO)` to indicate that this field represents the `IdentityObject` on the *to* side of the relationship.

The `IdentityObjectRelationshipType` entity is a lookup table containing the valid relationship types. The `@IdentityProperty(PropertyType.NAME)` annotation is used to indicate the field containing the relationship type names:

```java
@Entity
public class IdentityObjectRelationshipType implements Serializable {
  @Id @GeneratedValue private Long id;

  @IdentityProperty(PropertyType.NAME)
  private String name;

  // snip getter and setter methods
}
```

32.2.6. Attributes

The attribute table is used to store any additional information that is to be associated with identity objects. Here's an example of an entity bean used to store attributes:

```java
@Entity
@IdentityEntity(IDENTITY_ATTRIBUTE)
public class IdentityObjectAttribute implements Serializable {

  @Id @GeneratedValue private Integer attributeId;

  @ManyToOne
  @JoinColumn(name = "IDENTITY_OBJECT_ID")
  private IdentityObject identityObject;

  @IdentityProperty(PropertyType.NAME)
  private String name;

  @IdentityProperty(PropertyType.VALUE)
  private String value;

  // snip getter and setter methods
```

```
}
```

The `name` field is annotated with `@IdentityProperty(PropertyType.NAME)` to indicate that this field contains the attribute name. The `value` field is annotated with `@IdentityProperty(PropertyType.VALUE)` to indicate that this field contains the attribute value.

32.3. Managing Users, Groups and Roles

The Identity Management features are provided by a number of manager objects, which can be access from an `IdentitySession`. The `IdentitySession` may be injected directly into your beans like so:

```java
import org.picketlink.idm.api.IdentitySession;

public @Model class IdentityAction {
  @Inject IdentitySession identitySession;

  // code goes here...
}
```

Once you have the `IdentitySession` object, you can use it to perform various identity management operations. You should refer to the PicketLink documentation for a complete description of the available features, however the following sections contain a brief overview.

32.3.1. Managing Users and Groups

Users and groups are managed by a `PersistenceManager`, which can be obtained by calling `getPersistenceManager()` on the `IdentitySession` object:

```java
PersistenceManager pm = identitySession.getPersistenceManager();
```

Once you have the `PersistenceManager` object, you can create `User` objects with the `createUser()` method:

```java
User user = pm.createUser("john");
```

Similarly, you can create `Group` objects with the `createGroup()` method:

```java
Group headOffice = pm.createGroup("Head Office", "OFFICE");
```

You can also remove users and groups by calling the `removeUser()` or `removeGroup()` method.

32.3.2. Managing Relationships

Relationships are used to associate `User` objects with `Group` objects. Relationships can be managed with the `RelationshipManager` object, which can be obtained by calling `getRelationshipManager()` on the `IdentitySession`:

```
RelationshipManager rm = identitySession.getRelationshipManager();
```

Relationships are created by invoking the `associateUser()` method, and passing in the group and user objects that should be associated:

```
rm.associateUser(headOffice, user);
```

32.3.3. Managing Roles

Roles are managed via the `RoleManager` object, which can be obtained by invoke the `getRoleManager()` method on the `IdentitySession` object:

```
RoleManager roleManager = identitySession.getRoleManager();
```

Roles are an association between a user and a group, however they are slightly more complex than a simple group membership as the association also has a *role type*. The role type is generally used to describe a particular function of the user within the group. Role types are represented by the `RoleType` object, and can be created with the `createRoleType()` method:

```
RoleType manager = roleManager.createRoleType("manager");
```

Roles can be assigned to users by invoking the `createRole()` method, and passing in the `RoleType`, `User` and `Group`:

```
Role r = roleManager.createRole(manager, user, headOffice);
```

Security - External Authentication

33.1. Introduction

The external authentication module is an optional add-on to the core Seam Security module, which provides a number of features that enable your application to authenticate against third party identity services, via a number of supported protocols.

> ⚠️ **Warning**
>
> The features described in this chapter are a *preview* only. The APIs described may change in a subsequent version of Seam, and may not be backwards-compatible with previous versions.

Currently this module supports authentication via OpenID, and other protocols (such as SAML and OAuth) are currently under development for the next version of Seam.

33.1.1. Configuration

If your project is Maven-based, then add the following dependency to your project:

```
<dependency>
  <groupId>org.jboss.seam.security</groupId>
  <artifactId>seam-security-external</artifactId>
</dependency>
```

If you are not using Maven, you must add the `seam-security-external.jar` library to your project, which can be found in the Seam Security downloadable distribution.

33.2. OpenID

OpenID allows the users of your application to authenticate without requiring them to create an account. When using OpenID, your user is temporarily redirected to the web site of their OpenID provider so that they can enter their password, after which they are redirected back to your application. The OpenID authentication process is safe - at no time is the user's password seen by any site besides their OpenID provider.

33.2.1. Overview

The external authentication module provides support for OpenID based on *OpenID4Java* [http://code.google.com/p/openid4java/], an open source OpenID library (licensed under the Apache

v2 license) with both Relying Party and Identity Provider capabilities. This feature allows your application to authenticate its users against an external OpenID provider, such as Google or Yahoo, or to turn your application into an OpenID provider itself.

> **Note**
>
> To see the OpenID features in action, take a look at the `openid-rp` example included in the Seam Security distribution.

33.2.2. Enabling OpenID for your application

To use OpenID in your own application, you must configure Seam Security to use `OpenIdAuthenticator`, an `Authenticator` implementation that performs authentication against an OpenID provider. This authenticator is a named, session-scoped bean, with the following declaration:

```
public @Named("openIdAuthenticator") @SessionScoped class OpenIdAuthenticator
```

33.2.2.1. Using OpenID as your only authentication method

If your application only uses OpenID to provide authentication services, then it is recommended that `OpenIdAuthenticator` is selected by configuring the `authenticatorClass` property of the `Identity` bean. The following code sample demonstrates how this might be done by using Solder:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:s="urn:java:ee"
  xmlns:security="urn:java:org.jboss.seam.security"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee    http://jboss.org/schema/cdi/beans_1_0.xsd">

  <security:Identity>
    <s:modifies/>
    <security:authenticatorClass>org.jboss.seam.security.external.openid.OpenIdAuthenticator</security:authenticatorClass>
  </security:Identity>
```

33.2.2.2. Using OpenID as one of many possible authentication methods

If your application gives the user a choice of which authentication method to use, then it is not possible to pre-configure which `Authenticator` implementation is used to authenticate. In these circumstances, it is recommended that you configure the authenticator by specifying a value for the `authenticatorName` property of the `Identity` bean. This can be done by binding a view-layer control such as a radio group directly to this property, to allow the user to select the method of authentication they wish to use. See the following JSF code as an example:

```
<h:outputLabel value="Authenticate using:"/>
<h:selectOneRadio id="authenticator" value="#{identity.authenticatorName}">
 <f:selectItem itemLabel="OpenID" itemValue="openIdAuthenticator" />
 <f:selectItem itemLabel="Custom" itemValue="customAuthenticator" />
</h:selectOneRadio>
```

33.2.3. Choosing which OpenID provider to use

Seam provides built-in support for a number of well-known OpenID providers. The `OpenIdAuthenticator` bean may be configured to select which OpenID provider will be used to process an authentication request. Each concrete provider implements the following interface:

```
public interface OpenIdProvider {
   String getCode();
   String getName();
   String getUrl();
}
```

The following table lists the providers that come pre-packaged in Seam:

Provider	Code	Name	URL
CustomOpenIdProvider	custom	Google	
GoogleOpenIdProvider	google	Google	https://www.google.com/accounts/o8/id
MyOpenIdProvider	myopenid	MyOpenID	https://myopenid.com
YahooOpenIdProvider	yahoo	Yahoo	https://me.yahoo.com

To select one of the built-in providers to use for an authentication request, the `providerCode` property of the `OpenIdAuthenticator` bean should be set to one of the Code values from

the above table. The `OpenIdAuthenticator` bean provides a convenience method called `getProviders()` that returns a list of all known providers. This may be used in conjunction with a radio group to allow the user to select which OpenID provider they wish to authenticate with - see the following JSF snippet for an example:

```
<h:selectOneRadio value="#{openIdAuthenticator.providerCode}">
<f:selectItems value="#{openIdAuthenticator.providers}" var="p" itemValue="#{p.code}" itemLabel="#{p.name}"/>
>
</h:selectOneRadio>
```

33.2.3.1. Using a custom OpenID provider

If you would like to allow your users to specify an OpenID provider that is not supported out of the box by Seam, then the `CustomOpenIdProvider` may be used. As it is a `@Named` bean, it can be accessed directly from the view layer via EL. The following JSF code shows how you might allow the user to specify their own OpenID provider:

```
<h:outputLabel value="If you have selected the Custom OpenID provider, please provide a URL:"/
>
<h:inputText value="#{customOpenIdProvider.url}"/>
```

33.2.4. Managing the OpenID authentication process

Your application must provide an implementation of the `OpenIdRelyingPartySpi` interface to process OpenID callback events. This interface declares the following methods:

```
public interface OpenIdRelyingPartySpi {
    void loginSucceeded(OpenIdPrincipal principal, ResponseHolder responseHolder);
    void loginFailed(String message, ResponseHolder responseHolder);
```

The implementation is responsible for processing the response of the OpenID authentication, and is typically used to redirect the user to an appropriate page depending on whether authentication was successful or not.

There are two API calls that *must* be made in the case of a successful authentication. The first one should notify the `OpenIdAuthenticator` that the authentication attempt was successful, and pass it the `OpenIdPrincipal` object:

> **⚠ Warning**
>
> If the following two API calls are omitted, unpredictable results may occur!

```
openIdAuthenticator.success(principal);
```

Secondly, a `DeferredAuthenticationEvent` must be fired to signify that a deferred authentication attempt has been completed:

```
deferredAuthentication.fire(new DeferredAuthenticationEvent());
```

After making these two API calls, the implementation may perform whatever additional logic is required. The following code shows a complete example:

```java
import java.io.IOException;

import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.servlet.ServletContext;

import org.jboss.seam.security.events.DeferredAuthenticationEvent;
import org.jboss.seam.security.external.api.ResponseHolder;
import org.jboss.seam.security.external.openid.OpenIdAuthenticator;
import org.jboss.seam.security.external.openid.api.OpenIdPrincipal;
import org.jboss.seam.security.external.spi.OpenIdRelyingPartySpi;

public class OpenIdRelyingPartySpiImpl implements OpenIdRelyingPartySpi {
    @Inject private ServletContext servletContext;
    @Inject OpenIdAuthenticator openIdAuthenticator;
    @Inject Event<DeferredAuthenticationEvent> deferredAuthentication;

    public void loginSucceeded(OpenIdPrincipal principal, ResponseHolder responseHolder) {
        try {
            openIdAuthenticator.success(principal);
            deferredAuthentication.fire(new DeferredAuthenticationEvent());

            responseHolder.getResponse().sendRedirect(servletContext.getContextPath() + "/UserInfo.jsf");
        } catch (IOException e) {
```

```
        throw new RuntimeException(e);
    }
  }

  public void loginFailed(String message, ResponseHolder responseHolder) {
    try {
            responseHolder.getResponse().sendRedirect(servletContext.getContextPath()  +  "/
AuthenticationFailed.jsf");
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Security - Authorization

34.1. Configuration

Before using any of Seam's authorization features, you must enable the `SecurityInterceptor` by adding the following code to your application's `beans.xml`:

```
<interceptors>
  <class>org.jboss.seam.security.SecurityInterceptor</class>
</interceptors>
```

34.2. Basic Concepts

Seam Security provides a number of facilities for restricting access to certain parts of your application. As mentioned previously, the security API is centered around the `Identity` bean, which is a session-scoped bean used to represent the *identity* of the current user.

To be able to restrict the sensitive parts of your code, you may inject the `Identity` bean into your class:

```
@Inject Identity identity;
```

Once you have injected the `Identity` bean, you may invoke its methods to perform various types of authorization. The following sections will examine each of these in more detail.

The security model in Seam Security is based upon the PicketLink API. Let's briefly examine a few of the core interfaces provided by PicketLink that are used in Seam.
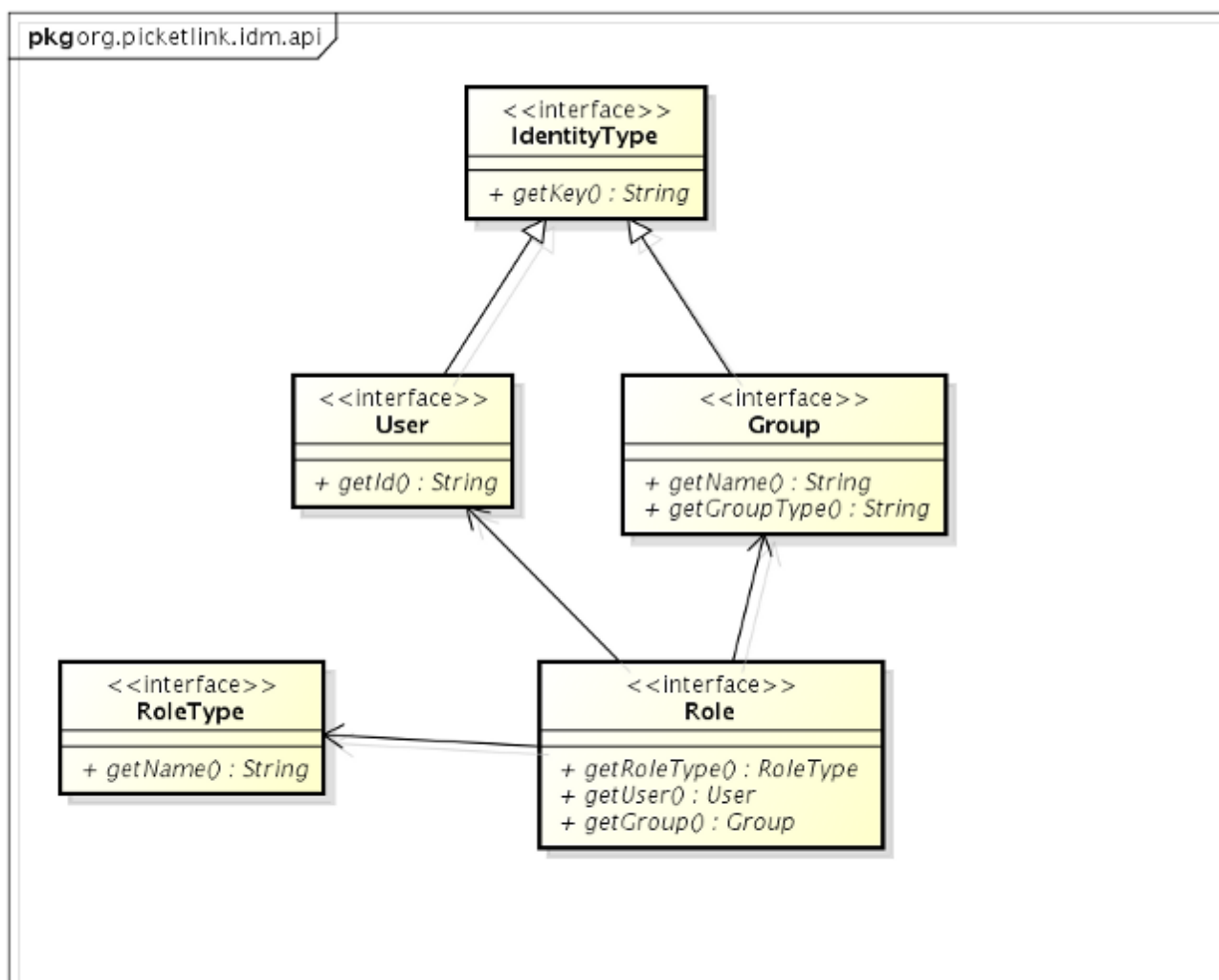
powered by astah*

34.2.1. IdentityType

This is the common base interface for both `User` and `Group`. The `getKey()` method should return a unique identifying value for the identity type.

34.2.2. User

Represents a user. The `getId()` method should return a unique value for each user.

34.2.3. Group

Represents a group. The `getName()` method should return the name of the group, while the `getGroupType()` method should return the group type.

34.2.4. Role

Represents a role, which is a direct one-to-one typed relationship between a User and a Group. The `getRoleType()` method should return the role type. The `getUser()` method should return

the User for which the role is assigned, and the `getGroup()` method should return the Group that the user is associated with.

34.2.5. RoleType

Represents a role type. The `getName()` method should return the name of the role type. Some examples of role types might be `admin`, `superuser`, `manager`, etc.

34.3. Role and Group-based authorization

This is the simplest type of authorization, used to define coarse-grained privileges for users assigned to a certain role or belonging to a certain group. Users may belong to zero or more roles and groups, and inversely, roles and groups may contain zero or more members.

> **Note**
>
> The concept of a *role* in Seam Security is based upon the model defined by PicketLink. I.e, a role is a direct relationship between a user and a group, which consists of three aspects - a member, a role name and a group (see the class diagram above). For example, user *Bob* (the member) may be an *admin* (the role name) user in the *HEAD OFFICE* group.

The `Identity` bean provides the following two methods for checking role membership:

```
boolean hasRole(String role, String group, String groupType);
void checkRole(String role, String group, String groupType);
```

These two methods are similar in function, and both accept the same parameter values. Their behaviour differs when an authorization check fails. The `hasRole()` returns a value of `false` when the current user is not a member of the specified role. The `checkRole()` method on the other hand, will throw an `AuthorizationException`. Which of the two methods you use will depend on your requirements.

The following code listing contains a usage example for the `hasRole()` method:

```
if (identity.hasRole("manager", "Head Office", "OFFICE")) {
    report.addManagementSummary();
}
```

Groups can be used to define a collection of users that meet some common criteria. For example, an application might use groups to define users in different geographical locations, their role in the company, their department or division or some other criteria which may be significant from

a security point of view. As can be seen in the above class diagram, groups consist of a unique combination of group name and group type. Some examples of group types may be "OFFICE", "DEPARTMENT", "SECURITY_LEVEL", etc. An individual user may belong to many different groups.

The `Identity` bean provides the following methods for checking group membership:

```
boolean inGroup(String name, String groupType);
void checkGroup(String group, String groupType);
```

These methods are similar in behaviour to the role-specific methods above. The `inGroup()` method returns a value of `false` when the current user isn't in the specified group, and the `checkGroup()` method will throw an exception.

34.4. Typesafe authorization

Seam Security provides a way to secure your bean classes and methods by annotating them with a *typesafe security binding*. Each security binding must have a matching authorizer method, which is responsible for performing the business logic required to determine whether a user has the necessary privileges to invoke a bean method. Creating and applying a security binding is quite simple, and is described in the following steps.

34.4.1. Creating a typesafe security binding

A typesafe security binding is an annotation, meta-annotated with the `SecurityBindingType` annotation:

```
import org.jboss.seam.security.annotations.SecurityBindingType;

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

The security binding annotation may also define member values, which are taken into account when matching the annotated bean class or method with an authorizer method. All member values are taken into consideration, except for those annotated with `@Nonbinding`, in much the same way as a qualifier binding type.

```
import javax.enterprise.util.Nonbinding;
import org.jboss.seam.security.annotations.SecurityBindingType;
```

```
@SecurityBindingType
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Foo {
   String bar();
   @Nonbinding String other() default "";
}
```

34.4.2. Creating an authorizer method

The next step after creating the security binding type is to create a matching authorizer method. This method must contain the business logic required to perform the required authorization check, and return a `boolean` value indicating whether the authorization check passed or failed.

An authorizer method must be annotated with the `@Secures` annotation, and the security binding types for which it is performing the authorization check. An authorizer method may declare zero or more method parameters. Any parameters defined by the authorizer method are treated as injection points, and are automatically injected by the Seam Security extension. The following example demonstrates an authorizer method that injects the `Identity` bean, which is then used to perform the authorization check.

```
import org.jboss.seam.security.annotations.Secures;

public class Restrictions {
  public @Secures @Admin boolean isAdmin(Identity identity) {
    return identity.hasRole("admin", "USERS", "GROUP");
  }
}
```

> **Note**
>
> Authorizer methods will generally make use of the security API to perform their security check, however this is not a hard restriction.

34.4.3. Applying the binding to your business methods

Once the security binding annotation and the matching authorizer method have been created, the security binding type may be applied to a bean class or method. If applied at the class level, every method of the bean class will have the security restriction applied. Methods annotated with a security binding type also inherit any security bindings on their declaring class. Both bean classes and methods may be annotated with multiple security bindings.

```
public @ConversationScoped class UserAction {
 public @Admin void deleteUser(String userId) {
   // code
 }
}
```

If a security check fails when invoking a method annotated with a security binding type, an `AuthorizationException` is thrown. Solder can be used to handle this exception gracefully, for example by redirecting them to an error page or displaying an error message. Here's an example of an exception handler that creates a JSF error message:

```
@HandlesExceptions
public class ExceptionHandler {
   @Inject FacesContext facesContext;
                          public        void        handleAuthorizationException(@Handles
 CaughtException<AuthorizationException> evt) {
     facesContext.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR,
        "You do not have the necessary permissions to perform that operation", ""));
     evt.handled();
  }
}
```

34.4.4. Built-in security binding annotations

Seam Security provides one security binding annotation out of the box, `@LoggedIn`. This annotation may be applied to a bean to restrict its methods to only those users that are currently authenticated.

```
import org.jboss.seam.security.annotations.LoggedIn;

public @LoggedIn class CustomerAction {
 public void createCustomer() {
   // code
 }
}
```

Security - Events

35.1. Introduction

A number of CDI events are fired during the course of many security-related operations, allowing additional business logic to be executed in response to certain security events. This is useful if you would like to generate additional logging or auditing, or produce messages to display to the user.

35.2. Event list

The following table contains the list of event classes that may be fired by Seam Security, along with a description of when the event is fired. All event classes are contained in the `org.jboss.seam.security.events` package.

Event	Description
AlreadyLoggedInEvent	Fired when a user who is already logged in attempts to log in again
AuthorizationCheckEvent	Fired when an authorization check is performed, such as `Identity.hasPermission().`
CredentialsUpdatedEvent	Fired whenever a user's credentials (such as their username or password) are updated.
DeferredAuthenticationEvent	Fired when a deferred authentication occurs. For example, at the end of the OpenID authentication process when the OpenID provider redirects the user back to the application.
LoggedInEvent	Fired when the user is successfully logged in.
LoginFailedEvent	Fired when an authentication attempt by the user fails.
NotAuthorizedEvent	Fired when the user is not authorized to invoke a particular operation.
NotLoggedInEvent	Fired when the user attempts to invoke a privileged operation before they have authenticated.
PreAuthenticateEvent	Fired just before a user is authenticated
PostAuthenticateEvent	Fired after a user has authenticated successfully.
PreLoggedOutEvent	Fired just before a user is logged out.
PostLoggedOutEvent	Fired after a user has logged out.

Event	Description
PrePersistUserEvent	Fired just before a new user is persisted (when using Identity Management).
PrePersistUserRoleEvent	Fired just before a new user role is persisted (when using Identity Management).
QuietLoginEvent	Fired when a user is quietly authenticated.
SessionInvalidatedEvent	Fired when a user's session is invalidated.
UserAuthenticatedEvent	Fired when a user is authenticated.
UserCreatedEvent	

35.3. Usage Example

The following code listing shows the `SecurityEventMessages` class, from the Seam Security implementation library. This class (which is disabled by default due to the `@Veto` annotation) uses the Messages API from Seam International to generate user-facing messages in response to certain security events.

```
package org.jboss.seam.security;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

import org.jboss.seam.international.status.Messages;
import org.jboss.seam.security.events.AlreadyLoggedInEvent;
import org.jboss.seam.security.events.LoggedInEvent;
import org.jboss.seam.security.events.LoginFailedEvent;
import org.jboss.seam.security.events.NotLoggedInEvent;
import org.jboss.seam.security.events.PostAuthenticateEvent;
import org.jboss.solder.core.Requires;
import org.jboss.solder.core.Veto;

public @ApplicationScoped @Veto @Requires("org.jboss.seam.international.status.Messages")
class SecurityEventMessages {
   private static final String DEFAULT_LOGIN_FAILED_MESSAGE = "Login failed - please check
 your username and password before trying again.";
   private static final String DEFAULT_LOGIN_SUCCESSFUL_MESSAGE = "Welcome, {0}.";
    private static final String DEFAULT_ALREADY_LOGGED_IN_MESSAGE = "You're already
 logged in. Please log out first if you wish to log in again.";
   private static final String DEFAULT_NOT_LOGGED_IN_MESSAGE = "Please log in first.";

   public void postAuthenticate(@Observes PostAuthenticateEvent event, Messages messages,
 Identity identity) {
```

```
      messages.info(DEFAULT_LOGIN_SUCCESSFUL_MESSAGE, identity.getUser().getId());
  }

     public void addLoginFailedMessage(@Observes LoginFailedEvent event, Messages
messages) {
     messages.error(DEFAULT_LOGIN_FAILED_MESSAGE);
  }

     public void addLoginSuccessMessage(@Observes LoggedInEvent event, Messages
messages, Credentials credentials) {
    messages.info(DEFAULT_LOGIN_SUCCESSFUL_MESSAGE, credentials.getUsername());
  }

 public void addAlreadyLoggedInMessage(@Observes AlreadyLoggedInEvent event, Messages
messages) {
     messages.error(DEFAULT_ALREADY_LOGGED_IN_MESSAGE);
  }

     public void addNotLoggedInMessage(@Observes NotLoggedInEvent event, Messages
messages) {
     messages.error(DEFAULT_NOT_LOGGED_IN_MESSAGE);
  }
}
```

Part V. Seam International

Introduction

The goal of Seam International is to provide a unified approach to configuring locale, timezone and language. With features such as Status message propagation to UI, multiple property storage implementations and more.

Installation

Most features of Seam International are installed automatically by including `seam-international.jar` in the web application library folder. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your `pom.xml` file:

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

> **Tip**
>
> Replace ${seam-international-version} with the most recent or appropriate version of Seam International.

Locales

37.1. Application Locale

In a similar fashion to TimeZones we have an Application `Locale`:

```
@Inject
private java.util.Locale lc;
```

accessible via EL with "defaultLocale".

By default the `Locale` will be set to the JVM default, unless you produce a String annotated with `@DefaultLocale`. This can be achieved through either the Seam Config module, with any bean that `@Produces` a method or field that matches the type and qualifier.

This will set the application language to be English with the country of US:

```
@Produces
@DefaultLocale
private String defaultLocaleKey = "en_US";
```

As you can see from the previous example, you can define the `Locale` with `lang_country_variant`. It's important to note that the first two parts of the locale definition are not expected to be greater than 2 characters otherwise an error will be produced and it will default to the JVM `Locale`.

37.2. User Locale

The Locale associated with the User Session can be retrieved by:

```
@Inject
@Client
private java.util.Locale locale;
```

which is EL accessible via `userLocale`.

By default the `Locale` will be that of the Application when the User Session is initialized. However, changing the User's `Locale` is a simple matter of firing an event to update it. An example would be:

```
@Inject
```

```
@Client
@Alter
private Event<java.util.Locale> localeEvent;

public void setUserLocale() {
   Locale canada = Locale.CANADA;
   localeEvent.fire(canada);
}
```

37.3. Available Locales

We've also provided a list of available Locales that can be accessed via:

```
@Inject
private List<java.util.Locale> locales;
```

The locales that will be returned as available can be defined by extending `LocaleConfiguration`.
As seen here:

```
public class CustomLocaleConfiguration extends LocaleConfiguration {
   @PostConstruct
   public void setup() {
      addSupportedLocaleKey("en");
      addSupportedLocaleKey("fr");
   }
}
```

Timezones

To support a more developer friendly way of handling TimeZones, in addition to supporting JDK `TimeZone`, we have added support for using Joda-Time through their `DateTimeZone` class. Don't worry, it provides convenience methods for converting to JDK `TimeZone`.

38.1. Joda Time

To activate Joda-Time for i18n within your project you will need to add the following Maven dependency:

```xml
<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>1.6</version>
  </dependency>
```

38.2. Application TimeZone

We have an Application time zone that is available with Joda-Time (`DateTimeZone`) or the JDK (`TimeZone`) that can be retrieved with

```java
@Inject
private DateTimeZone applicationDateTimeZone;

@Inject
private TimeZone applicationTimeZone
```

It can also be accessed through EL by the name "defaultDateTimeZone" for Joda-Time or "defaultTimeZone" for JDK!

By default the `TimeZone` will be set to the JVM default, unless you produce a String annotated with `@DefaultTimeZone`. This can be achieved through either the Seam Config module or any bean that `@Produces` a method or field that matches the type and qualifier.

This will set the application time zone to be Tijuana:

```java
@Produces
@DefaultTimeZone
private String defaultTimeZoneId = "America/Tijuana";
```

38.3. User TimeZone

In addition to the Application time zone, there is also a time zone assigned to each User Session.

```
@Inject
@Client
private DateTimeZone userDateTimeZone;

@Inject
@Client
private TimeZone userTimeZone;
```

It can also be accessed through EL using "userDateTimeZone" for Joda-Time and "userTimeZone" for JDK.

By default the `DateTimeZone` and `TimeZone` for a User Session is initialized to the same as the Application. However, changing the User's `DateTimeZone` and `TimeZone` is a simple matter of firing an event to update it. An example would be

```
@Inject
@Client
@Alter
private Event<DateTimeZone> dtzEvent;

@Inject
@Client
@Alter
private Event<TimeZone> tzEvent;

public void setUserDateTimeZone() {
    DateTimeZone dtzTijuana = DateTimeZone.forID("America/Tijuana");
    dtzEvent.fire(dtzTijuana);

    TimeZone tzTijuana = TimeZone.getTimeZone("America/Tijuana");
    tzEvent.fire(tzTijuana);
}
```

38.4. Available TimeZones

We've also provided a list of available TimeZones that can be accessed via

```
@Inject
private List<ForwardingDateTimeZone> dateTimeZones;

@Inject
private List<ForwardingTimeZone> timeZones;
```

191

Messages

39.1. Message Creation

There are currently two ways to create a message within the module.

The first would mostly be used when you don't want to add the generated message directly to the UI, but want to log it out, or store it somewhere else

```java
@Inject
private MessageFactory factory;

public String getMessage() {
    MessageBuilder builder = factory.info("There are {0} cars, and they are all {1}; {1} is the best color.", 5, "green");
    return builder.build().getText();
}
```

The second is to add the message to a list that will be returned to the UI for display.

```java
@Inject
private Messages messages;

public void setMessage() {
    messages.info("There are {0} cars, and they are all {1}; {1} is the best color.", 5, "green");
}
```

Either of these methods supports the four message levels which are info, warning, error and fatal.

Both MessageFactory and Messages support four ways in which to create a Message:

- Directly adding the message

- Directly adding the message and replacing parameters

- Retrieving the message from a bundle

- Retrieving the message from a bundle and replacing parameters

Examples for each of these are:

```java
messages.info("Simple Text");
```

```
messages.info("Simple Text with {0} parameter", 1);
```

```
messages.info(new BundleKey("org.jboss.international.seam.test.TestBundle", "key1"));
```

```
messages.info(new BundleKey("org.jboss.international.seam.test.TestBundle", "key2"), 1);
```

39.2. Properties Files

The examples in the previous section on how to create a message from a properties file made the assumption that you had already created it! Now we tell you how to actually do that.

When creating a `BundleKey` in the previous section, we were passing it a bundle name of "org.jboss.international.seam.test.TestBundle". This name is essentially the path to the properties file! Let me explain. As we all know properties files need to be on the classpath for our code to find them, so "org.jboss.international.seam.test.TestBundle" is telling our code that on the classpath there is a `TestBundle.properties` file located at a path of `org/jboss/international/seam/test`.

To create a property file for another language, it's simply a case of appending the name of the locale to the end of the file name. Such as `TestBundle_fr.properties` for French or `TestBundle_en_US.properties` for American English.

> **Note**
>
> If you only ever intend to use a single language within your application, there is no need to create a locale specific properties file, as the non locale version will be used if a locale specific properties file is not present.

Part VI. Seam Faces

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.

40.1. 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 inadvertently 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!*

40.2. 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!

40.3. 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 because Seam Faces scans for the presence of the Seam Servlet, and programatically registers it for you if it's not present.

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.

41.1. 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.

41.1.1. 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:

```java
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:

```java
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:

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

41.1.2. 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.

41.2. 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.

41.2.1. 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
}
```

41.2.2. 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

41.2.3. 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 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).

42.1. @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 `RenderContext`s 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.

42.2. @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;
    // ...
}
```

42.3. @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`.

43.1. 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.

```java
@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.

```java
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?"

43.2. 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
<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.

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

44.1. @*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.

```java
@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

```java
@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();
```

```
   }
}
```

44.2. @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;`
	`}`

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.

45.1. 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
<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`

45.2. <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.

```html
<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."));
    }
  }
}
```

<s:validateForm>

> **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 ValidatorException
    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));
    }
    ...
  }
```

45.3. <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.

45.3.1. 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.

45.3.2. 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.

45.3.3. 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.

45.4. 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 (_).

Part VII. Seam Reports

Introduction

The goal of Seam Reports is to provide a fully integrated CDI programming model portable extension for Java EE that provides APIs for compiling, populating and rendering reports from existing report frameworks.

Seam Reports contains similar functionality to that of the Excel and PDF templates of Seam 2, however, the creation and compilation of the reports is quite different. Seam Reports aligns much better with existing tools in a business, making use of knowledge and expertise that exists outside of development. The functionality in Seam 2 was largely targeted to creating flyers and simple pages. Seam Reports can accomplish this, but also allows for easy creation of multi-page business reports by integrating with *JasperReports* [http://jasperforge.org/projects/jasperreports], *Pentaho* [http://www.pentaho.com/], and *XDocReports* [http://code.google.com/p/xdocreport/]. Integration with other reporting solutions can also be done easily by implementing five small interfaces provided by Seam Reports, see chapter 3 for more information about adding reporting engines.

Installation

Most features of Seam Reports are installed automatically by including the seam-reports-api.jar and the respective provider implementation (along with its dependencies) in the web application library folder. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your pom.xml file:

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

<!-- If you are using Jasper Reports, add the following dependency -->
<dependency>
    <groupId>org.jboss.seam.reports</groupId>
    <artifactId>seam-reports-jasper</artifactId>
    <version>${seam-reports-version}</version>
</dependency>

<!-- If you are using Pentaho, add the following dependency -->
<dependency>
    <groupId>org.jboss.seam.reports</groupId>
    <artifactId>seam-reports-pentaho</artifactId>
    <version>${seam-reports-version}</version>
</dependency>
```

> **Tip**
>
> Replace ${seam-reports-version} with the most recent or appropriate version of Seam Reports.

46.1. Installation using Seam Forge

If you are using Seam Forge, you may use the seam-reports plugin to help with the setup.

46.1.1. Plugin Installation

If not already installed yet on Forge, you may install the plugin by running the following command inside Forge:

```
forge git-plugin git://github.com/forge/plugin-seam-reports.git
```

46.1.2. Plugin Configuration

• To add only the api:

```
seam-reports setup
```

• To configure Seam Reports to work with JasperReports:

```
seam-reports setup --provider JASPER
```

• To configure Seam Reports to work with Pentaho Reporting Engine:

```
seam-reports setup --provider PENTAHO
```

Usage

47.1. Quick Start

Using Seam Reports is a simple four step process. These steps are the same regardless of the reporting engine being used.

1. Create a report using a favorite report editor

2. Load the created report

3. Fill the report with data

4. Render the report

Of course some of these steps will have different ways of accomplishing the task, but at a high level they are all the same. For simplicity this quick start will use JasperReports and the first step will be assumed to have already taken place and the report is available in the deployed archive. The location of the report isn't important, the ability to pull it into an `InputStream` is all that really matters.

The following code demonstrates a basic way of fulfilling the last three steps in using Seam Reports using JasperReports as the reporting engine. The report has already been created and is bundled inside the deployable archive. There are no paramaters for the report. The report is a simple listing of people's names and contact information.

```
@Model
public class PersonContactReport {
  @Inject @Resource("WEB-INF/jasperreports/personContact.jrxml")    ①
  private InputStream reportTemplate;

  @Inject @Jasper                                                    ②
  private ReportCompiler reportCompiler;

  @Inject @Jasper @PDF                                               ③
  private ReportRenderer pdfRenderer;

  @Inject
  private EntityManager em;
                                                                     ④
  public OutputStream render() {
    final Report filledReport = this.fillReport();
    final OutputStream os = new ByteArrayOutputStream();
```

```
  this.pdfRenderer.render(filledReport, os);
  return os;
 }
```

⑤

```
 private Report fillReport() {
  final ReportDefinition rd = this.reportCompiler.compile(reportTemplate);
  return rd.fill(this.createDatasource(), Collections.EMPTY_MAP);
 }
```

⑥

```
 private JRDataSource createDatasource() {
          final    List<Person>    personList    =    this.em.createQuery("select    p    from
 Person", Person.class).getResultList();
  return new JRBeanCollectionDataSource(personList);
 }
}
```

① Solder allows easy resource injection for files available in the archive. This injects the report template which has been created previously (perhaps by someone else in the business) and added to the deployable archive.

② A `ReportCompiler` is an interface from Seam Reports which abstracts compiling the report template into `ReportDefinition`. Seam Reports makes use of CDI's type safety features by using qualifiers to further narrow the intended type. This allows programs to remain implementation agnostic. The `Jasper` qualifier annotation instructs CDI to inject an implementation of the `ReportCompiler` which contains the same qualifer.

③ This is an instance of using both a qualifer (`@Jasper`) and also a metadata annotation, which happens to be a stereotype. The `@PDF` annotation is a CDI stereotype, which essentially means it's a group of other annotations. It carries metadata about the type it is decorating. More about this later.

④ The `render` method is the only entry point into the class, it also returns the final output of generating a report. It makes use of other methods in the class to finish the steps outlined above to generate a report using Seam Reports.

⑤ At this stage data to populate the report is retrieved and added to the compiled `ReportDefinition`. This particular report doesn't make use of any parameters, hence the empty map instance being passed.

⑥ This last stage of using Seam Reports is the only place that may require the application to use the report engine API. In this example a list of JPA entities is retrieved and added to a JasperReports datasource, which is then used by the calling method to populate the report template as mentioned above.

47.2. Annotations

There are four API level annotations to be aware of when using Seam Reports. All four of them declare metadata about objects that are being injected. They're also all CDI stereotypes which instruct the implementing renderer the mimetype that should be used.

- CSV

- PDF

- XLS

- XML

These annotations are only used when injecting a `ReportRenderer`. Only one of them may be used per renderer. Multiple renderers must be injected if multiple renderering types are desired.

47.3. Troubleshooting

Part VIII. Seam Mail

Seam Mail Introduction

Seam mail is an portable CDI extension designed to make working with Java Mail easier via standard methods or `plugable` templating engines.

48.1. Getting Started

No better way to start off then with a simple example to show what we are talking about.

```java
@Inject
private Instance<MailMessage> mailMessage;

public void sendMail() {

    MailMessage m = mailMessage.get();
    m.from("John Doe<john@test.com>")
        .to("Jane Doe<jane@test.com>")
        .subject(subject)
        .bodyHtml(htmlBody)
        .importance(MessagePriority.HIGH)
        .send();
}
```

Very little is required to enable this level of functionality in your application. Let's start off with a little required configuration.

Configuration

By default the configuration parameters for Seam Mail are handled via configuration read from your application's seam-beans.xml. This file is then parsed by Seam Solder to configure the MailConfig class. You can override this and provide your own configuration outside of Seam Mail but we will get into that later.

49.1. Minimal Configuration

First lets add the relevant maven configuration to your pom.xml

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

Now now that is out of the way lets provide `JavaMail` with the details of your `SMTP` server so that it can connect and send your mail on it's way.

This configuration is handled via Seam Solder which reads in the configuration from your application's `seam-beans.xml` and configures the `MailConfig` class prior to injection.

```xml
<beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:java:ee"
    xmlns:mail="urn:java:org.jboss.seam.mail.core"
    xsi:schemaLocation="
   http://java.sun.com/xml/ns/javaee
   http://docs.jboss.org/cdi/beans_1_0.xsd">

  <mail:MailConfig
       serverHost="my-server.test.com"
       serverPort="25">
     <s:modifies/>
  </mail:MailConfig>

</beans>
```

That is all the configuration necessary to send a simple email message. Next we will take a look at how to configure and use the supported templating engines.

> **Important**
>
> JBoss AS 7.0.x does not correctly load all the modules to support sending mail *AS7-1375* [https://issues.jboss.org/browse/AS7-1375]. This is easily fixed By replacing the module definition at $JBOSS_HOME/modules/javax/activation/api/main/module.xml with the following
>
> ```xml
> <module xmlns="urn:jboss:module:1.0" name="javax.activation.api">
>   <dependencies>
>     <module name="javax.api" />
>     <module name="javax.mail.api" >
>        <imports><include path="META-INF"/></imports>
>     </module>
>   </dependencies>
>
>   <resources>
>     <resource-root path="activation-1.1.1.jar"/>
>
>     <!-- Insert resources here -->
>   </resources>
> </module>
> ```
>
> This will be fixed in AS 7.1.x

Core Usage

50.1. Intro

While Seam Mail does provide methods to produce templated email, there is a core set of functionality that is shared whether you use a templating engine or not.

50.2. Contacts

At it's base an email consists of various destinations and content. Seam Mail provides a wide varerity of methods of ways to configure the following address fields

- From

- To

- CC

- BCC

- REPLY-TO

50.2.1. String Based

Seam Mail leverages the JavaMail InternetAddress object internally for parsing and storage and provides a varargs method for each of the contact types. Thus you can provide either a String, multiple Strings or a String []. Addresses are parsed as RFC 822 addresses and can be a valid Email Address or a Name + Email Address.

```
MailMessage m = mailMessage.get();
m.from("John Doe<john@test.com>")
.to("jane@test.com")
.cc("Dan<dan@test.com>", "bill@test.com")
```

50.2.2. InternetAddress

Since we leverage standard InternetAddress object we might as well provide a method to use it.

```
MailMessage m = mailMessage.get();
m.from(new InternetAddress("John Doe<john@test.com>"))
```

50.2.3. EmailContact

Since applications frequently have their own object to represent a user who will have an email set to them we provide a simple interface which your object can implement.

```java
public interface EmailContact {
  public String getName();

  public String getAddress();
}
```

Let's define this interface on an example user entity

```java
@Entity
public class User implements EmailContact {

  private String username; //"john@test.com"
  private String firstName; //"John"
  private String lastName; //"Doe"

  public String getName() {
    return firstName + " " + lastName;
  }

  public String getAddress() {
    return username;
  }
}
```

Now we can use our User object directly in an of the contact methods

```java
User user;

MailMessage m = mailMessage.get();
m.from("John Doe<john@test.com>")
 .to(user)
```

50.2.4. Content

TODO

50.2.5. Attachments

TODO

Templating

51.1. Velocity

TO DO

51.2. Freemarker

TO DO

Advanced Features

52.1. MailTransporter

TO DO

52.2. MailConfig

TO DO

Part IX. Seam Remoting

Seam Remoting - Basic Features

Seam provides a convenient method of remotely accessing CDI beans from a web page, using AJAX (Asynchronous Javascript and XML). The framework for this functionality is provided with almost no up-front development effort - your beans only require simple annotating to become accessible via AJAX. This chapter describes the steps required to build an AJAX-enabled web page, then goes on to explain the features of the Seam Remoting framework in more detail.

53.1. Configuration

To use remoting, the Seam Remoting servlet must first be configured in your `web.xml` file:

```xml
<servlet>
  <servlet-name>Remoting Servlet</servlet-name>
  <servlet-class>org.jboss.seam.remoting.Remoting</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Remoting Servlet</servlet-name>
  <url-pattern>/seam/resource/remoting/*</url-pattern>
</servlet-mapping>
```

> **Note**
>
> If your application is running within a Servlet 3.0 (or greater) environment, then the servlet configuration listed above is not necessary as the Seam Remoting JAR library bundles a `web-fragment.xml` that configures the Remoting servlet automatically.

The next step is to import the necessary Javascript into your web page. There are a minimum of two scripts that must be imported. The first one contains all the client-side framework code that enables remoting functionality:

```html
<script type="text/javascript" src="seam/resource/remoting/resource/remote.js"></script>
```

By default, the client-side JavaScript is served in compressed form, with white space compacted and JavaScript comments removed. For a development environment, you may wish to use the uncompressed version of `remote.js` for debugging and testing purposes. To do this, simply add the `compress=false` parameter to the end of the url:

```
<script          type="text/javascript"          src="seam/resource/remoting/resource/remote.js?
compress=false"></script>
```

The second script that you need contains the stubs and type definitions for the beans you wish
to call. It is generated dynamically based on the method signatures of your beans, and includes
type definitions for all of the classes that can be used to call its remotable methods. The name of
the script reflects the name of your bean. For example, if you have a named bean annotated with
@Named, then your script tag should look like this (for a bean class called CustomerAction):

```
<script type="text/javascript"
       src="seam/resource/remoting/interface.js?customerAction"></script>
```

Otherwise, you can simply specify the fully qualified class name of the bean:

```
<script type="text/javascript"
       src="seam/resource/remoting/interface.js?com.acme.myapp.CustomerAction"></script>
```

If you wish to access more than one bean from the same page, then include them all as parameters
of your script tag:

```
<script type="text/javascript"
       src="seam/resource/remoting/interface.js?customerAction&accountAction"></script>
```

53.1.1. Dynamic type loading

If you forget to import a bean or other class that is required by your bean, don't worry. Seam
Remoting has a dynamic type loading feature that automatically loads any JavaScript stubs for
bean types that it doesn't recognize.

53.2. The "Seam" object

Client-side interaction with your beans is all performed via the Seam Javascript object. This object
is defined in remote.js, and you'll be using it to make asynchronous calls against your bean.
It contains methods for creating client-side bean objects and also methods for executing remote
requests. The easiest way to become familiar with this object is to start with a simple example.

53.2.1. A Hello World example

Let's step through a simple example to see how the Seam object works. First of all, let's create a
new bean called helloAction:

```
@Named
public class HelloAction implements HelloLocal {
   @WebRemote public String sayHello(String name) {
      return "Hello, " + name;
   }
}
```

Take note of the @WebRemote annotation on the sayHello() method in the above listing. This annotation makes the method accessible via the Remoting API. Besides this annotation, there's nothing else required on your bean to enable it for remoting.

> **Note**
>
> If you are performing a persistence operation in the method marked @WebRemote you will also need to add a @Transactional annotation to the method. Otherwise, your method would execute outside of a transaction without this extra hint.That's because unlike a JSF request, Seam does not wrap the remoting request in a transaction automatically.

Now for our web page - create a new JSF page and import the helloAction bean:

```
<script type="text/javascript"
      src="seam/resource/remoting/interface.js?helloAction
```

To make this a fully interactive user experience, let's add a button to our page:

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

We'll also need to add some more script to make our button actually do something when it's clicked:

```
<script type="text/javascript">
 //<![CDATA[

 function sayHello() {
  var name = prompt("What is your name?");
  Seam.createBean("helloAction").sayHello(name, sayHelloCallback);
 }

 function sayHelloCallback(result) {
```

```
  alert(result);
 }


  // ]]>
</script>
```

We're done! Deploy your application and open the page in a web browser. Click the button, and enter a name when prompted. A message box will display the hello message confirming that the call was successful. If you want to save some time, you'll find the full source code for this Hello World example in the `/examples/helloworld` directory.

So what does the code of our script actually do? Let's break it down into smaller pieces. To start with, you can see from the Javascript code listing that we have implemented two methods - the first method is responsible for prompting the user for their name and then making a remote request. Take a look at the following line:

Seam.createBean("helloAction").sayHello(name, sayHelloCallback);

The first section of this line, `Seam.createBean("helloAction")` returns a proxy, or "stub" for our `helloAction` bean. We can invoke the methods of our bean against this stub, which is exactly what happens with the remainder of the line: `sayHello(name, sayHelloCallback);`.

What this line of code in its completeness does, is invoke the `sayHello` method of our bean, passing in `name` as a parameter. The second parameter, `sayHelloCallback` isn't a parameter of our bean's `sayHello` method, instead it tells the Seam Remoting framework that once it receives the response to our request, it should pass it to the `sayHelloCallback` Javascript method. This callback parameter is entirely optional, so feel free to leave it out if you're calling a method with a `void` return type or if you don't care about the result.

The `sayHelloCallback` method, once receiving the response to our remote request then pops up an alert message displaying the result of our method call.

53.2.2. Seam.createBean

The `Seam.createBean` JavaScript method is used to create client-side instances of both action and "state" beans. For action beans (which are those that contain one or more methods annotated with `@WebRemote`), the stub object provides all of the remotable methods exposed by the bean. For "state" beans (i.e. beans that simply carry state, for example Entity beans) the stub object provides all the same accessible properties as its server-side equivalent. Each property also has a corresponding getter/setter method so you can work with the object in JavaScript in much the same way as you would in Java.

53.3. The Context

The Seam Remoting Context contains additional information which is sent and received as part of a remoting request/response cycle. It currently contains the conversation ID and Call ID, and may be expanded to include other properties in the future.

53.3.1. Setting and reading the Conversation ID

If you intend on using remote calls within the scope of a conversation then you need to be able to read or set the conversation ID in the Seam Remoting Context. To read the conversation ID after making a remote request call `Seam.context.getConversationId()`. To set the conversation ID before making a request, call `Seam.context.setConversationId()`.

If the conversation ID hasn't been explicitly set with `Seam.context.setConversationId()`, then it will be automatically assigned the first valid conversation ID that is returned by any remoting call. If you are working with multiple conversations within your page, then you may need to explicitly set the conversation ID before each call. If you are working with just a single conversation, then you don't need to do anything special.

53.3.2. Remote calls within the current conversation scope

In some circumstances it may be required to make a remote call within the scope of the current view's conversation. To do this, you must explicitly set the conversation ID to that of the view before making the remote call. This small snippet of JavaScript will set the conversation ID that is used for remoting calls to the current view's conversation ID:

```
Seam.context.setConversationId( #{conversation.id} );
```

53.4. Working with Data types

53.4.1. Primitives / Basic Types

This section describes the support for basic data types. On the server side these values as a rule are compatible with either their primitive type or their corresponding wrapper class.

53.4.1.1. String

Simply use Javascript String objects when setting String parameter values.

53.4.1.2. Number

There is support for all number types supported by Java. On the client side, number values are always serialized as their String representation and then on the server side they are converted to the correct destination type. Conversion into either a primitive or wrapper type is supported for `Byte`, `Double`, `Float`, `Integer`, `Long` and `Short` types.

53.4.1.3. Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

53.4.2. JavaBeans

In general these will be either entity beans or JavaBean classes, or some other non-bean class. Use `Seam.createBean()` to create a new instance of the object.

53.4.3. Dates and Times

Date values are serialized into a String representation that is accurate to the millisecond. On the client side, use a JavaScript `Date` object to work with date values. On the server side, use any `java.util.Date` (or descendent, such as `java.sql.Date` or `java.sql.Timestamp` class.

53.4.4. Enums

On the client side, enums are treated the same as `String`s. When setting the value for an enum parameter, simply use the `String` representation of the enum. Take the following bean as an example:

```java
@Named
public class paintAction {
  public enum Color {red, green, blue, yellow, orange, purple};

  public void paint(Color color) {
    // code
  }
}
```

To call the `paint()` method with the color `red`, pass the parameter value as a `String` literal:

```java
Seam.createBean("paintAction").paint("red");
```

The inverse is also true - that is, if a bean method returns an enum parameter (or contains an enum field anywhere in the returned object graph) then on the client-side it will be converted to a `String`.

53.4.5. Collections

53.4.5.1. Bags

Bags cover all collection types including arrays, collections, lists, sets, (but excluding Maps - see the next section for those), and are implemented client-side as a JavaScript array. When calling

a bean method that accepts one of these types as a parameter, your parameter should be a JavaScript array. If a bean method returns one of these types, then the return value will also be a JavaScript array. The remoting framework is clever enough on the server side to convert the bag to an appropriate type (including sophisticated support for generics) for the bean method call.

53.4.5.2. Maps

As there is no native support for Maps within JavaScript, a simple Map implementation is provided with the Seam Remoting framework. To create a Map which can be used as a parameter to a remote call, create a new `Seam.Map` object:

```
var map = new Seam.Map();
```

This JavaScript implementation provides basic methods for working with Maps: `size()`, `isEmpty()`, `keySet()`, `values()`, `get(key)`, `put(key,  value)`, `remove(key)` and `contains(key)`. Each of these methods are equivalent to their Java counterpart. Where the method returns a collection, such as `keySet()` and `values()`, a JavaScript Array object will be returned that contains the key or value objects (respectively).

53.5. Debugging

To aid in tracking down bugs, it is possible to enable a debug mode which will display the contents of all the packets send back and forth between the client and server in a popup window. To enable debug mode, set the `Seam.debug` property to `true` in Javascript:

```
Seam.debug = true;
```

If you want to write your own messages to the debug log, call `Seam.log(message).`

53.6. Messages

The Seam International module provides a Messages API that allows generation of view-independent messages. This is useful if you want to convey additional information to a user that is not returned directly from the result of a method invocation.

Using the Messages API is extremely easy. Simply add the Seam International libraries to your application (see the Seam International configuration chapter to learn how to do this), then inject the `Messages` object into your bean. The `Messages` object provides several methods for adding messages, see the Seam International documentation for more information. Here's a simple example showing how to create an `info` message (messages generally follow the same DEBUG, INFO, WARN, ERROR levels that a typical logging framework would provide):

```
import javax.inject.Inject;
```

```java
import org.jboss.seam.international.status.Messages;
import org.jboss.seam.remoting.annotations.WebRemote;

public class HelloAction {
  @Inject Messages messages;

  @WebRemote
  public String sayHello(String name) {
    messages.info("Invoked HelloAction.sayHello()");
    return "Hello, " + name;
  }
}
```

After creating the message in your server-side code, you still need to write some client-side code to handle any messages that are returned by your remote invocations. Thankfully this is also simple, you just need to write a JavaScript handler function and assign it to `Seam.messageHandler`.

If any messages are returned from a remote method invocation, the message handler function will be invoked and passed a list of Message objects. These objects declare three methods for retrieving various properties of the message - `getLevel()` returns the message level (such as DEBUG, INFO, etc). The `getTargets()` method returns the targets of the message - these may be the ID's for specific user interface controls, which is helpful for conveying validation failures for certain field values. The `getTargets()` method may return null, if the message is not specific to any field value. Lastly, the `getText()` method returns the actual text of the message.

Here's a really simple example showing how you would display an alert box for any messages returned:

```javascript
function handleMessages(msgs) {
 for (var i = 0; i < msgs.length; i++) {
   alert("Received message - Level: " + msgs[i].getLevel() + " Text: " + msgs[i].getText();
 }
}

Seam.messageHandler = handleMessages;
```

You can see the Messages API in action in the HelloWorld example. Simply choose the "Formal" option for the Formality, and "Localized (English)" for the Localization. Invoking this combination will cause a server-side message to be created, which you will then see in the Messages list at the top of the screen.

53.7. Handling Exceptions

When invoking a remote bean method, it is possible to specify an exception handler which will process the response in the event of an exception during bean invocation. To specify an exception handler function, include a reference to it after the callback parameter in your JavaScript:

```
var callback = function(result) { alert(result); };
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.createBean("helloAction").sayHello(name, callback, exceptionHandler);
```

If you do not have a callback handler defined, you must specify `null` in its place:

```
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.createBean("helloAction").sayHello(name, null, exceptionHandler);
```

The exception object that is passed to the exception handler exposes two methods, `getExceptionClass()` which returns the name of the exception class that was thrown, and `getMessage()`, which returns the exception message which is produced by the exception thrown by the `@WebRemote` method.

It is also possible to register a global exception handler, which will be invoked if there is no exception handler defined for an individual invocation. By default, the global exception handler will display an alert message notifying the user that there was an exception - here's what the default exception handler looks like:

```
Seam.defaultExceptionHandler = function(exception) {
    alert("An exception has occurred while executing a remote request: " +
 exception.getExceptionClass() + ":" + exception.getMessage());
};
```

If you would like to provide your own global exception handler, then simply override the value of `Seam.exceptionHandler` with your own custom exception handler, as in the following example:

```
 function customExceptionHandler(exception) {
    alert("Uh oh, something bad has happened! [" + exception.getExceptionClass() + ":" +
 exception.getMessage() + "]");
 }

 Seam.exceptionHandler = customExceptionHandler;
```

53.8. The Loading Message

The default loading message that appears in the top right corner of the screen can be modified, its rendering customised or even turned off completely.

53.8.1. Changing the message

To change the message from the default "Please Wait..." to something different, set the value of `Seam.loadingMessage`:

```
Seam.loadingMessage = "Loading...";
```

53.8.2. Hiding the loading message

To completely suppress the display of the loading message, override the implementation of `displayLoadingMessage()` and `hideLoadingMessage()` with functions that instead do nothing:

```
// don't display the loading indicator
Seam.displayLoadingMessage = function() {};
Seam.hideLoadingMessage = function() {};
```

53.8.3. A Custom Loading Indicator

It is also possible to override the loading indicator to display an animated icon, or anything else that you want. To do this override the `displayLoadingMessage()` and `hideLoadingMessage()` messages with your own implementation:

```
Seam.displayLoadingMessage = function() {
  // Write code here to display the indicator
};

Seam.hideLoadingMessage = function() {
  // Write code here to hide the indicator
};
```

53.9. Controlling what data is returned

When a remote method is executed, the result is serialized into an XML response that is returned to the client. This response is then unmarshaled by the client into a JavaScript object. For complex types (i.e. Javabeans) that include references to other objects, all of these referenced objects are also serialized as part of the response. These objects may reference other objects, which may reference other objects, and so forth. If left unchecked, this object "graph" could potentially be enormous, depending on what relationships exist between your objects. And as a side issue (besides the potential verbosity of the response), you might also wish to prevent sensitive information from being exposed to the client.

Seam Remoting provides a simple means to "constrain" the object graph, by specifying the `exclude` field of the remote method's `@WebRemote` annotation. This field accepts a String array containing one or more paths specified using dot notation. When invoking a remote method, the objects in the result's object graph that match these paths are excluded from the serialized result packet.

For all our examples, we'll use the following `Widget` class:

```java
public class Widget
{
  private String value;
  private String secret;
  private Widget child;
  private Map<String,Widget> widgetMap;
  private List<Widget> widgetList;

  // getters and setters for all fields
}
```

53.9.1. Constraining normal fields

If your remote method returns an instance of `Widget`, but you don't want to expose the `secret` field because it contains sensitive information, you would constrain it like this:

```java
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

The value "secret" refers to the `secret` field of the returned object. Now, suppose that we don't care about exposing this particular field to the client. Instead, notice that the `Widget` value that is returned has a field `child` that is also a `Widget`. What if we want to hide the `child`'s `secret` value instead? We can do this by using dot notation to specify this field's path within the result's object graph:

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

53.9.2. Constraining Maps and Collections

The other place that objects can exist within an object graph are within a `Map` or some kind of collection (`List`, `Set`, `Array`, etc). Collections are easy, and are treated like any other field. For example, if our `Widget` contained a list of other `Widget`s in its `widgetList` field, to constrain the `secret` field of the `Widget`s in this list the annotation would look like this:

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

To constrain a `Map`'s key or value, the notation is slightly different. Appending `[key]` after the `Map`'s field name will constrain the `Map`'s key object values, while `[value]` will constrain the value object values. The following example demonstrates how the values of the `widgetMap` field have their `secret` field constrained:

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

53.9.3. Constraining objects of a specific type

There is one last notation that can be used to constrain the fields of a type of object no matter where in the result's object graph it appears. This notation uses either the name of the bean (if the object is a named bean) or the fully qualified class name (only if the object is not a named bean) and is expressed using square brackets:

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

53.9.4. Combining Constraints

Constraints can also be combined, to filter objects from multiple paths within the object graph:

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

Seam Remoting - Model API

54.1. Introduction

The Model API builds on top of Seam Remoting's object serialization features to provide a *component-based* approach to working with a server-side object model, as opposed to the *RPC-based* approach provided by the standard Remoting API. This allows a client-side representation of a server-side object graph to be modified ad hoc by the client, after which the changes made to the objects in the graph can be *applied* to the corresponding server-side objects. When applying the changes the client determines exactly which objects have been modified by recursively walking the client-side object tree and generating a delta by comparing the original property values of the objects with their new property values.

This approach, when used in conjunction with the extended persistence context provided by Seam elegantly solves a number of problems faced by AJAX developers when working remotely with persistent objects. A persistent, managed object graph can be loaded at the start of a new conversation, and then across multiple requests the client can fetch the objects, make incremental changes to them and apply those changes to the same managed objects after which the transaction can be committed, thereby persisting the changes made.

One other useful feature of the Model API is its ability to *expand* a model. For example, if you are working with entities with lazy-loaded associations it is usually not a good idea to blindly fetch the associated objects (which may in turn themselves contain associations to other entities, ad nauseum), as you may inadvertently end up fetching the bulk of your database. Seam Remoting already knows how to deal with lazy-loaded associations by automatically excluding them when marshalling instances of entity beans, and assigning them a client-side value of `undefined` (which is a special JavaScript value, distinct from `null`). The Model API goes one step further by giving the client the option of manipulating the associated objects also. By providing an *expand* operation, it allows for the initialization of a previously-uninitialized object property (such as a lazy-loaded collection), by dynamically "grafting" the initialized value onto the object graph. By *expanding* the model in this way, we have at our disposal a powerful tool for building dynamic client interfaces.

54.2. Model Operations

For the methods of the Model API that accept action parameters, an instance of `Seam.Action` should be used. The constructor for `Seam.Action` takes no parameters:

```
var action = new Seam.Action();
```

The following table lists the methods used to define the action. Each of the following methods return a reference to the `Seam.Action` object, so methods can be chained.

Table 54.1. Seam.Action method reference

Method	Description
`setBeanType(beanType)`	Sets the class name of the bean to be invoked. • `beanType` - the fully qualified class name of the bean type to be invoked.
`setQualifiers(qualifiers)`	Sets the qualifiers for the bean to be invoked. • `qualifiers` - a comma-separated list of bean qualifier names. The names may either be the simple or fully qualified names of the qualifier classes.
`setMethod(method)`	Sets the name of the bean method. • `method` - the name of the bean method to invoke.
`addParam(param)`	Adds a parameter value for the action method. This method should be called once for each parameter value to be added, in the correct parameter order. • `param` - the parameter value to add.

The following table describes the methods provided by the `Seam.Model` object. To work with the Model API in JavaScript you must first create a new Model object:

```
var model = new Seam.Model();
```

Table 54.2. Seam.Model method reference

Method	Description
`addBean(alias, bean, qualifiers)`	Adds a bean value to the model. When the model is fetched, the value of the specified bean will be read and placed into the model, where it may be accessed by using the `getValue()` method with the specified alias. Can only be used before the model is fetched. • `alias` - the local alias for the bean value. • `bean` - the name of the bean, either specified by the `@Named` annotation or the fully qualified class name. • `qualifiers` (optional) - a list of bean qualifiers.

Method	Description
`addBeanProperty(alias, bean, property, qualifiers)`	Adds a bean property value to the model. When the model is fetched, the value of the specified property on the specified bean will be read and placed into the model, where it may be accessed by using the `getValue()` method with the specified alias. Can only be used before the model is fetched. Example: ```\naddBeanProperty("account", "AccountAction",\n"account", "@Qualifier1", "@Qualifier2");\n``` • `alias` - the local alias for the bean value. • `bean` - the name of the bean, either specified by the `@Named` annotation or the fully qualified class name. • `property` - the name of the bean property. • `qualifiers` (optional) - a list of bean qualifiers. This parameter (and any after it) are treated as bean qualifiers.
`fetch(action, callback)`	Fetches the model - this operation causes an asynchronous request to be sent to the server. The request contains a list of the beans and bean properties (set by calling the `addBean()` and `addBeanProperty()` methods) for which values will be returned. Once the response is received, the callback method (if specified) will be invoked, passing in a reference to the model as a parameter. A model should only be fetched once. • `action` (optional) - a `Seam.Action` instance representing the bean action to invoke before the model values are read and stored in the model. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the model has been fetched. A reference to the model instance is passed to the callback method as a parameter.

Method	Description
`getValue(alias)`	This method returns the value of the object with the specified alias. • `alias` - the alias of the value to return.
`expand(value, property, callback)`	Expands the model by initializing a property value that was previously uninitialized. This operation causes an asynchronous request to be sent to the server, where the uninitialized property value (such as a lazy-loaded collection within an entity bean association) is initialized and the resulting value is returned to the client. Once the response is received, the callback method (if specified) will be invoked, passing in a reference to the model as a parameter. • `value` - a reference to the value containing the uninitialized property to fetch. This can be any value within the model, and does not need to be a "root" value (i.e. it doesn't need to be a value specified by `addBean()` or `addBeanProperty()`, it can exist anywhere within the object graph. • `property` - the name of the uninitialized property to be initialized. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the model has been expanded. A reference to the model instance is passed to the callback method as a parameter.
`applyUpdates(action, callback)`	Applies the changes made to the objects contained in the model. This method causes an asynchronous request to be sent to the server containing a delta consisting of a list of the changes made to the client-side objects. • `action` (optional) - a `Seam.Action` instance representing a bean method to be invoked after the client-side model changes have been applied to their corresponding server-side objects. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the updates have been applied. A reference to the model instance is passed to the callback method as a parameter.

54.3. Fetching a model

To fetch a model, one or more values must first be specified using `addBean()` or `addBeanProperty()` before invoking the `fetch()` operation. Let's work through an example - here we have an entity bean called `Customer`:

```java
@Entity Customer implements Serializable {
  private Integer customerId;
  private String firstName;
  private String lastName;

  @Id @GeneratedValue public Integer getCustomerId() { return customerId; }
  public void setCustomerId(Integer customerId) { this.customerId = customerId; }

  public String getFirstName() { return firstName; }
  public void setFirstName(String firstName) { this.firstName = firstName; }

  public String getLastName() { return lastName; }
  public void setLastName(String lastName) { this.lastName = lastName; }
}
```

We also have a bean called `CustomerAction`, which is responsible for creating and editing `Customer` instances. Since we're only interested in editing a customer right now, the following code only shows the `editCustomer()` method:

```java
@ConversationScoped @Named
public class CustomerAction {
  @Inject Conversation conversation;
  @PersistenceContext EntityManager entityManager;
  public Customer customer;

  public void editCustomer(Integer customerId) {
    conversation.begin();
    customer = entityManager.find(Customer.class, customerId);
  }

  public void saveCustomer() {
    entityManager.merge(customer);
    conversation.end();
  }
}
```

In the client section of this example, we wish to make changes to an existing `Customer` instance, so we need to use the `editCustomer()` method of `CustomerAction` to first load the customer entity, after which we can access it via the public `customer` field. Our model object must therefore be configured to fetch the `CustomerAction.customer` property, and to invoke the `editCustomer()` method when the model is fetched. We start by using the `addBeanProperty()` method to add a bean property to the model:

```
var model = new Seam.Model();
model.addBeanProperty("customer", "CustomerAction", "customer");
```

The first parameter of `addBeanProperty()` is the *alias* (in this case `customer`), which is used to access the value via the `getValue()` method. The `addBeanProperty()` and `addBean()` methods can be called multiple times to bind multiple values to the model. An important thing to note is that the values may come from multiple server-side beans, they aren't all required to come from the same bean.

CLIENT

SERVER

Model

Values

customer

Customer

+customerId: Integer
+firstName: String
+lastName: String

(Entity Bean)

CustomerAction

-conversation: Conversation
-entityManager: EntityManager
+customer: Customer

+createCustomer(): void
+editCustomer(customer:Custome
+saveCustomer(): void

(Conversation-scoped action
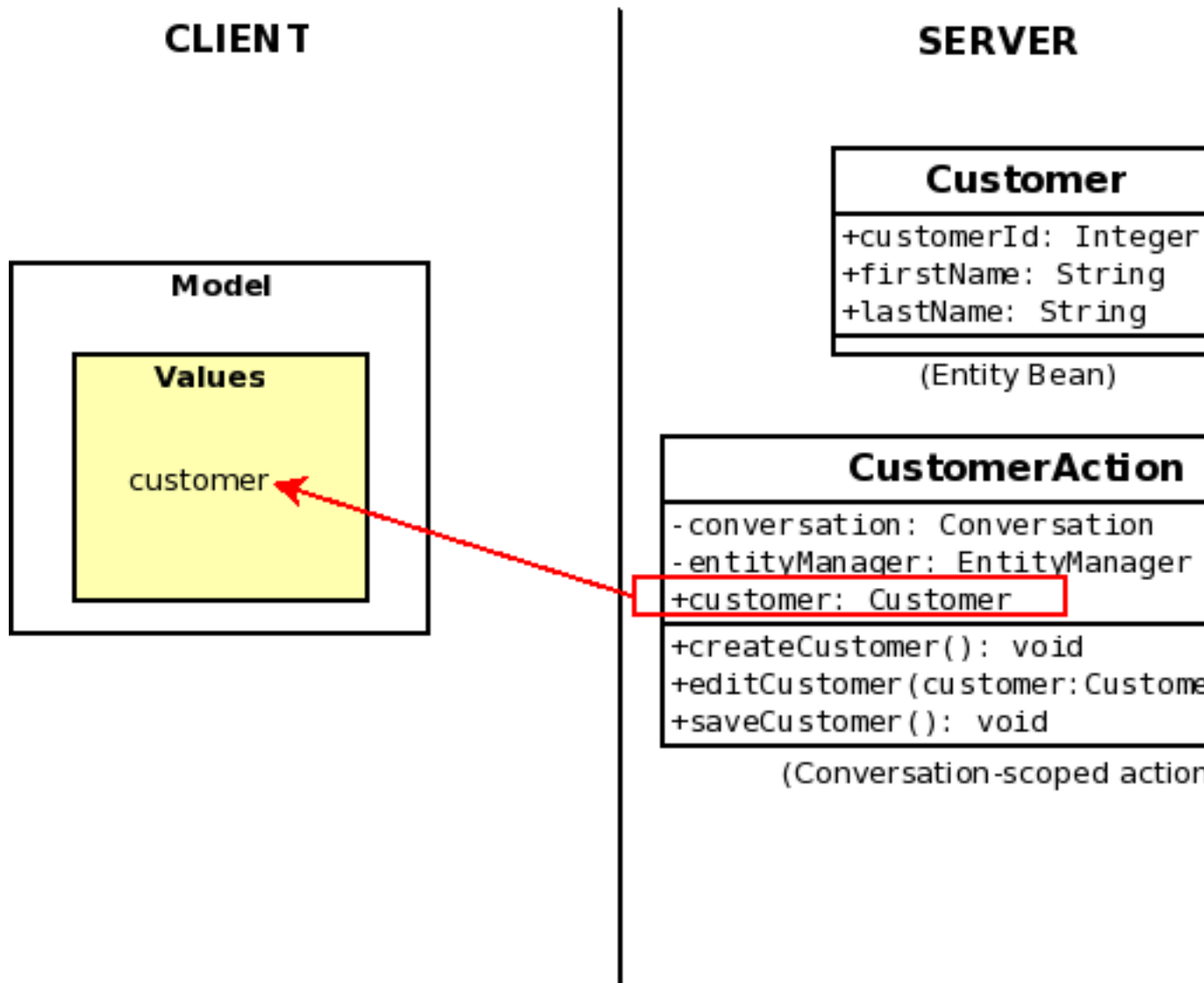
We also specify the action that we wish to invoke (i.e. the `editCustomer()` method). In this example we know the value of the `customerId` that we wish to edit, so we can specify this value as an action method parameter:

```
var action = new Seam.Action()
  .setBeanType("CustomerAction")
  .setMethod("editCustomer")
  .addParam(123);
```

Once we've specified the bean properties we wish to fetch and the action to invoke, we can then fetch the model. We pass in a reference to the action object as the first parameter of the `fetch()` method. Also, since this is an asynchronous request we need to provide a callback method to deal with the response. The callback method is passed a reference to the model object as a parameter.

```
        var     callback    =    function(model)    {    alert("Fetched    customer:    "
model.getValue("customer").firstName +
  " " + model.getValue("customer").lastName); };
model.fetch(action, callback);
```

When the server receives a model fetch request, it first invokes the action (if one is specified) before reading the requested property values and returning them to the client.

54.3.1. Fetching a bean value

Alternatively, if you don't wish to fetch a bean *property* but rather a bean itself (such as a value created by a producer method) then the `addBean()` method is used instead. Let's say we have a producer method that returns a qualified `UserSettings` value:

```
@Produces @ConversationScoped @Settings UserSettings getUserSettings() {
  /* snip code */
}
```

We would add this value to our model with the following code:

```
model.addBean("settings", "UserSettings", "@Settings");
```

The first parameter is the local alias for the value, the second parameter is the fully qualified class of the bean, and the third (and subsequent) parameter/s are optional bean qualifiers.

54.4. Modifying model values

Once a model has been fetched its values may be read using the `getValue()` method. Continuing on with the previous example, we would retrieve the `Customer` object via it's local alias (`customer`) like this:

```
var customer = model.getValue("customer");
```

We are then free to read or modify the properties of the value (or any of the other values within its object graph).

```
alert("Customer name is: " + customer.firstName + " " + customer.lastName);
customer.setLastName("Jones"); // was Smith, but Peggy got married on the weekend
```

54.5. Expanding a model

We can use the Model API's ability to expand a model to load uninitialized branches of the objects in the model's object graph. To understand how this works exactly, let's flesh out our example a little more by adding an `Address` entity class, and creating a one-to-many relationship between `Customer` and `Address`.
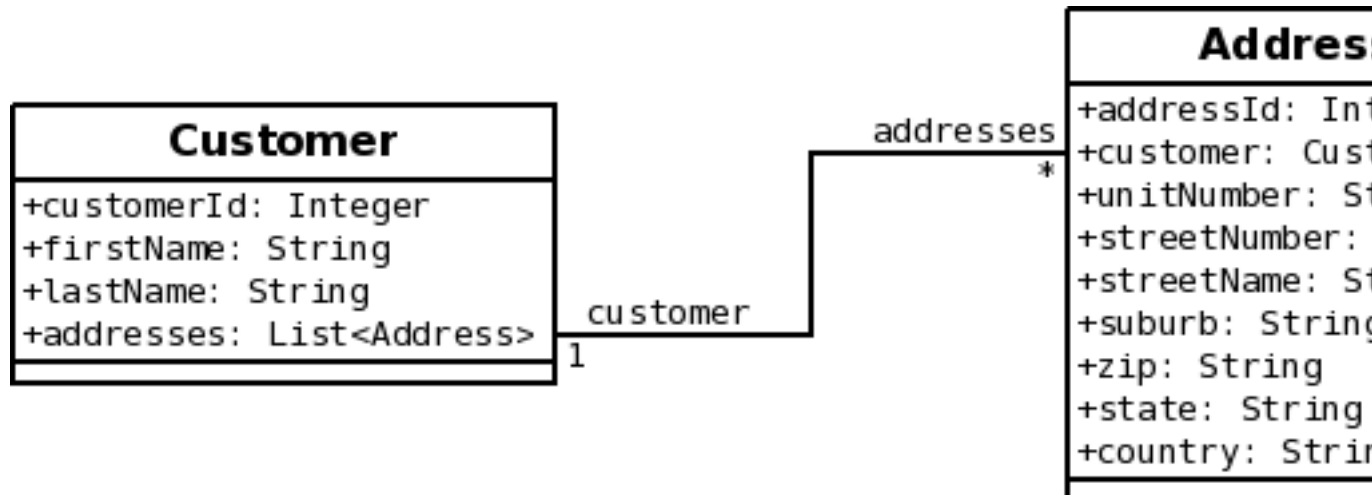

```
@Entity Address implements Serializable {
  private Integer addressId;
  private Customer customer;
  private String unitNumber;
  private String streetNumber;
  private String streetName;
  private String suburb;
  private String zip;
  private String state;
  private String country;

  @Id @GeneratedValue public Integer getAddressId() {  return addressId; }
  public void setAddressId(Integer addressId) { this.addressId = addressId; }

  @ManyToOne public Customer getCustomer() { return customer; }
  public void setCustomer(Customer customer) { this.customer = customer; }

  /* Snipped other getter/setter methods */

}
```

Here's the new field and methods that we also need to add to the `Customer` class:

```
private Collection<Address> addresses;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "customer", cascade = CascadeType.ALL)
public Collection<Address> getAddresses() { return addresses; }
public void setAddresses(Collection<Address> addresses) { this.addresses = addresses; }
```

As we can see, the `@OneToMany` annotation on the `getAddresses()` method specifies a `fetch` attribute of `LAZY`, meaning that by default the customer's addresses won't be loaded automatically when the customer is. When reading the *uninitialized* `addresses` property value from a newly-fetched `Customer` object in JavaScript, a value of `undefined` will be returned.

```
getValue("customer").addresses == undefined; // returns true
```

We can *expand* the model by making a special request to initialize this uninitialized property value. The `expand()` operation takes three parameters - the value containing the property to be initialized, the name of the property and an optional callback method. The following example shows us how the customer's `addresses` property can be initialized:

```
model.expand(model.getValue("customer"), "addresses");
```

The `expand()` operation makes an asynchronous request to the server, where the property value is initialized and the value returned to the client. When the client receives the response, it reads the initialized value and appends it to the model.

```
// The addresses property now contains an array of address objects
alert(model.getValue("customer").addresses.length + " addresses loaded");
```

54.6. Applying Changes

Once you have finished making changes to the values in the model, you can apply them with the `applyUpdates()` method. This method scans all of the objects in the model, compares them with their original values and generates a delta which may contain one or more changesets to send to the server. A changeset is simply a list of property value changes for a single object.

Like the `fetch()` command you can also specify an action to invoke when applying updates, although the action is invoked *after* the model updates have been applied. In a typical situation the invoked action would do things like commit a database transaction, end the current conversation, etc.

Since the `applyUpdates()` method sends an asynchronous request like the `fetch()` and `expand()` methods, we also need to specify a callback function if we wish to do something when the operation completes.

```
var action = new Seam.Action();
  .setBeanType("CustomerAction")
  .setMethod("saveCustomer");

var callback = function() { alert("Customer saved."); };

model.applyUpdates(action, callback);
```

The `applyUpdates()` method performs a refresh of the model, retrieving the latest state of the objects contained in the model after all updates have been applied and the action method (if specified) invoked.

Seam Remoting - Bean Validation

Seam Remoting provides integrated support for JSR-303 Bean Validation, which defines a standard approach for validating Java Beans no matter where they are used; web tier or persistence tier, server or client. Bean validation for remoting delivers JSR-303's vision by making all of the validation constraints declared by the server-side beans available on the client side, and allows developers to perform client-side bean validation in an easy to use, consistent fashion.

Client-side validation by its very nature is an asynchronous operation, as it is possible that the client may encounter a custom validation constraint for which it has no knowledge of the corresponding validation logic. Under these circumstances, the client will make a request to the server for the validation to be performed server-side, after which it receives the result will forward it to the client-side callback method. All built-in validation types defined by the JSR-303 specification are executed client-side without requiring a round-trip to the server. It is also possible to provide the client-side validation API with custom JavaScript to allow client-side execution of custom validations.

55.1. Validating a single object

The `Seam.validateBean()` method may be used to validate a single object. It accepts the following parameter values:

```
Seam.validateBean(bean, callback, groups);
```

The `bean` parameter is the object to validate.

The `callback` parameter should contain a reference to the callback method to invoke once validation is complete.

The `groups` parameter is optional, however may be specified if only certain validation groups should be validated. The `groups` parameter may be a `String` or an array of `String` values for when multiple groups are to be validated.

Here's an example showing how a bean called `customer` is validated:

```
function test() {
  var customer = Seam.createBean("com.acme.model.Customer");
  customer.setFirstName("John");
  customer.setLastName("Smith");
  Seam.validateBean(customer, validationCallback);
}

function validationCallback(violations) {
```

```
  if (violations.length == 0) alert("All validations passed!");
}
```

> **Tip**
>
> By default, when Seam Remoting performs validation for a single bean it will traverse the entire object graph for that bean and validate each unique object that it finds. If you don't wish to validate the entire object graph, then please refer to the section on validating multiple objects later in this chapter for an alternative.

55.2. Validating a single property

Sometimes it might not be desirable to perform validation for all properties of a bean. For example, you might have a dynamic form which displays validation errors as the user tabs between fields. In this situation, you may use the `Seam.validateProperty()` method to validate a single bean property.

```
Seam.validateProperty(bean, property, callback, groups)
```

The `bean` parameter is the object containing the property that is to be validated.

The `property` parameter is the name of the property to validate.

The `callback` parameter is a reference to the callback function to invoke once the property has been validated.

The `groups` parameter is optional, however may be specified if validating the property against a certain validation group. The `groups` parameter may be a `String` or an array of `String` values for multiple groups.

Here's an example showing how to validate the `firstName` property of a bean called `customer`:

```
function test() {
  var customer = Seam.createBean("com.acme.model.Customer");
  customer.setFirstName("John");
  Seam.validateProperty(customer, "firstName", validationCallback);
}

function validationCallback(violations) {
  if (violations.length == 0) alert("All validations passed!");
}
```

55.3. Validating multiple objects and/or properties

It is also possible to perform multiple validations for beans and bean properties in one go. This might be useful for example to perform validation of forms that present data from more than one bean. The `Seam.validate()` method takes the following parameters:

```
Seam.validate(validations, callback, groups);
```

The `validations` parameter should contain a list of the validations to perform. It may either be an associative array (for a single validation), or an array of associative arrays (for multiple validations) which define the validations that should be performed. We'll look at this parameter more closely in just a moment.

The `callback` parameter should contain a reference to the callback function to invoke once validation is complete. The optional `groups` parameter should contain the group name/s for which to perform validation.

The `groups` parameter allows one or more validation groups (specified by providing a `String` or array of `String` values) to be validated. The validation groups specified here will be applied to all bean values contained in the `validations` parameter.

The simplest example, in which we wish to validate a single object would look like this:

```
Seam.validate({bean:customer}, callback);
```

In the above example, validation will be performed for the `customer` object, after which the function named `validationCallback` will be invoked.

Validate multiple beans is done by passing in an array of validations:

```
Seam.validate([{bean:customer}, {bean:order}], callback);
```

Single properties can be validated by specifying a `property` name:

```
Seam.validate({bean:customer, property: "firstName"}, callback);
```

To prevent the entire object graph from being validated, the `traverse` property may be set to `false`:

```
Seam.validate({bean:customer, traverse: false}, callback);
```

Validation groups may also be set for each individual validation, by setting the `groups` property to a `String` or array of `String`s value:

```
Seam.validate({bean:customer, groups: "default"}, callback);
```

55.4. Validation groups

Validation group names should be the unqualified class name of the group class. For example, for the class `com.acme.InternalRegistration`, the client-side group name should be specified as `InternalRegistration`:

```
Seam.validateBean(user, callback, "InternalRegistration"
```

It is also possible to set the default validation groups against which all validations will be performed, by setting the `Seam.ValidationGroups` property:

```
Seam.ValidationGroups = ["Default", "ExternalRegistration"];
```

If no explicit group is set for the default, and no group is specified when performing validation, then the validation process will be executed against the 'Default' group.

55.5. Handling validation failures

If any validations fail during the validation process, then the callback method specified in the validation function will be invoked with an array of constraint violations. If all validations pass, this array will be empty. Each object in the array represents a single constraint violation, and contains the following property values:

`bean` - the bean object for which the validation failed.

`property` - the name of the property that failed validation

`value` - the value of the property that failed validation

`message` - a message string describing the nature of the validation failure

The callback method should contain business logic that will process the constraint violations and update the user interface accordingly to inform the user that validation has failed. The following

minimalistic example demonstrates how the validation errors can be displayed to the user as popup alerts:

```
function validationCallback(violations) {
for (var i = 0; i < violations.length; i++) {
  alert(violations[i].property + "=" + violations[i].value + " [violation] -> " + violations[i].message);
```

Part X. Seam REST

Introduction

Seam REST is a lightweight module that provides additional integration of technologies within the Java EE platform as well as third party technologies.

Seam REST is independent from CDI and JAX-RS implementations and thus fully portable between Java EE 6 environments.

Installation

The Seam REST module runs only on Java EE 6 compliant servers such as *JBoss Application Server* [http://www.jboss.org/jbossas] or *GlassFish* [https://glassfish.dev.java.net/] .

56.1. Basics

To use the Seam REST module, add `seam-rest` and `seam-rest-api` jars into the web application. If using Maven, add the following dependency into the web application's `pom.xml` configuration file.

Example 56.1. Dependency added to pom.xml

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

> **Tip**
>
> Substitute the expression ${seam.rest.version} with the most recent or appropriate version of Seam REST. Alternatively, you can create a *Maven user-defined property* [http://www.sonatype.com/books/mvnref-book/reference/resource-filtering-sect-properties.html#resource-filtering-sect-user-defined] to satisfy this substitution so you can centrally manage the version.

56.2. Transitive dependencies

Besides, Seam REST has several transitive dependencies (which are added automatically when using maven). Refer to *Section 61.1, "Transitive Dependencies"* for more details.

56.3. Registering JAX-RS components explicitly

The Seam REST module registers `SeamExceptionMapper` to hook into the exception processing mechanism of JAX-RS and `TemplatingMessageBodyWriter` to provide templating support.

These components are registered by default if classpath scanning of JAX-RS resources and providers is enabled (an empty `javax.ws.rs.core.Application` subclass is provided).

```java
@ApplicationPath("/api/*")
```

```
public class MyApplication extends Application {}
```

Otherwise, if the Application's `getClasses()` method is overridden to select resources and providers explicitly add `SeamExceptionMapper` and `TemplatingMessageBodyWriter`.

```java
@ApplicationPath("/api/*")
public class MyApplication extends Application
{
  @Override
  public Set<Class<?>> getClasses()
  {
    Set<Class<?>> classes = new HashSet<Class<?>>();
    ...
    ...
    ...
    classes.add(SeamExceptionMapper.class);
    classes.add(TemplatingMessageBodyWriter.class);
    return classes;
  }
}
```

56.4. Servlet container support

Seam REST can be used with plain Servlet containers such as Apache Tomcat 7. Firstly, we need to enhance the Servlet container capabilities. This is done by bundling Weld and RESTEasy within the application and configuring them. See the jaxrs-exceptions example and its `tomcat` build profile for more details.

In a EE6-compliant environment, Seam REST would be bootstrapped by a Servlet listener. However, `weld-servlet` does not support CDI injection into Servlet listeners. Therefore, add the following line to your application's `web.xml` file to bootstrap Seam REST using Servlet.

```xml
<servlet>
    <display-name>Servlet REST Listener Startup</display-name>
    <servlet-class>org.jboss.seam.rest.SeamRestStartupListener</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Exception Handling

The JAX-RS specification defines the mechanism for exception mapping providers as the standard mechanism for Java exception handling. The Seam REST module comes with an alternative approach, which is more consistent with the CDI programming model. It is also easier to use and still remains portable.

The Seam REST module allows you to:

- integrate with Solder exception handling framework and thus handle exceptions that occur in different parts of an application uniformly;

- define exception handling rules declaratively with annotations or XML.

57.1. Solder Exception Handling Integration

Solder exception handling framework handles exceptions within the Seam REST module: as result, an exception that occurs during an invocation of a JAX-RS service is routed through the Solder exception handling mechanism similar to the CDI event bus. This allows you to implement the exception handling logic in a loosely-coupled fashion.

The following code sample demonstrates a simple exception handler that converts the `NoResultException` exception to a 404 HTTP response.

Example 57.1. Solder Integration - `NoResultException` handler

```
@HandlesExceptions                                              ①
public class ExceptionHandler
{
  @Inject @RestResource
  ResponseBuilder builder                                       ②

  public void handleException(@Handles @RestRequest CaughtException<NoResultException> event)  ③
  {
    builder.status(404).entity("The requested resource does not exist.");
  }
}
```

① The `@HandlesExceptions` annotation marks the `ExceptionHandler` bean as capable of handling exceptions.

② The `ResponseBuilder` for creating the HTTP response is injected.

③ A method for handling `NoResultException` instances. Note that the `ExceptionHandler` can define multiple exception handling methods for various exception types.

Similarly to the CDI event bus, exceptions handled by a handler method can be filtered by qualifiers. The example above treats only exceptions that occur in a JAX-RS service invocation (as opposed to all exceptions of the given type that occur in the application, for example in the view layer). Thus, the `@RestRequest` qualifier is used to enable the handler only for exceptions that occur during JAX-RS service invocation.

For more information on Solder exception handling, refer to *Solder reference documentation* [http://docs.jboss.org/seam/3/solder/snapshot/reference/en-US/html/part5.html] .

57.2. Declarative Exception Mapping

Exception-mapping rules are often fairly simple. Thus, instead of being implemented programmatically, they can be expressed declaratively through metadata such as Java annotations or XML. The Seam REST module supports both ways of declarative configurations.

For each exception type, you can specify a status code and an error message of the HTTP response.

57.2.1. Annotation-based configuration

You can configure Seam REST exception mapping directly in your Java code with Java Annotations. An exception mapping rule is defined as a `@ExceptionMapping` annotation. Use an `@ExceptionMapping.List` annotation to define multiple exception mappings.

Example 57.2. Annotation-based exception mapping configuration

```
@ExceptionMapping.List({
@ExceptionMapping(exceptionType=NoResultException.class,status=404,message="Requested
 resource does not exist."),
@ExceptionMapping(exceptionType=IllegalArgumentException.class,status=400,message="Illegal
 argument value.")
})
@ApplicationPath("/api")
public MyApplication extends Application {
```

The `@ExceptionMapping` annotation can be applied on any Java class in the deployment. However, it is recommended to keep all exception mapping declarations in the same place, for example, in the `javax.ws.rs.core.Application` subclass.

Table 57.1. `@ExceptionMapping` properties

Name	Required	Default value	Description
exceptionType	true	-	Fully-qualified class name of the exception class

Name	Required	Default value	Description
status	true	-	HTTP status code
message	false	-	Error message sent within the HTTP response
useExceptionMessage	false	false	Exception error message
interpolateMessageBody	false	true	Enabling/disabling the EL interpolation of the error message
useJaxb	false	true	Enabling/disabling wrapping of the error message within a JAXB object. This allows marshalling to various media formats such as `application/xml`, `application/json`, etc.

57.2.2. XML configuration

As an alternative to the annotation-based configuration, you can use the configuration facilities provided by Solder to configure the `SeamRestConfiguration` class in XML.

For more information on how to use Solder for configuring beans, refer to the *Solder reference documentation* [http://docs.jboss.org/seam/3/solder/snapshot/reference/en-US/html/part4.html] . Once you have added the Seam XML module, specify the configuration in the `seam-beans.xml` file, located in the `WEB-INF` folder of the web archive.

Example 57.3. Exception mapping configuration in seam-beans.xml

```xml
<rest:SeamRestConfiguration>
  <rest:mappings>
    <s:value>

 <rest:Mapping exceptionType="javax.persistence.NoResultException" statusCode="404">
       <rest:message>Requested resource does not exist.</rest:message>
      </rest:Mapping>
    </s:value>
    <s:value>
     <rest:Mapping exceptionType="java.lang.IllegalArgumentException" statusCode="400">
```

```
            <rest:message>Illegal value.</rest:message>
        </rest:Mapping>
      </s:value>
  </rest:mappings>
</rest:SeamRestConfiguration>
```

Furthermore, you can use EL expressions in message templates to provide dynamic and more descriptive error messages.

Example 57.4. Exception mapping configuration in seam-beans.xml

```
<rest:Mapping exceptionType="javax.persistence.NoResultException" statusCode="404">
   <rest:message>Requested resource (#{uriInfo.path}) does not exist.</rest:message>
</rest:Mapping>
```

57.2.3. Declarative exception mapping processing

When an exception occurs at runtime, the `SeamExceptionMapper` first looks for a matching exception mapping rule. If it finds one, it creates an HTTP response with the specified status code and error message.

The error message is marshalled within a JAXB object and is thus available in multiple media formats. The most commonly used formats are XML and JSON. Most JAX-RS implementations provide media providers for both of these formats. In addition, the error message is also available in plain text.

Example 57.5. Sample HTTP response

```
HTTP/1.1 404 Not Found
Content-Type: application/xml
Content-Length: 123

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error>
   <message>Requested resource does not exist.</message>
</error>
```

Bean Validation Integration

Bean Validation (JSR-303) is a specification introduced as a part of Java EE 6. It aims to provide a standardized way of validating the domain model across all application layers.

The Seam REST module follows the Bean Validation specification and the incoming HTTP requests can be validated with this standardized mechanism.

58.1. Validating HTTP requests

Firstly, enable the `ValidationInterceptor` in the `beans.xml` configuration file.

```xml
<interceptors>
    <class>org.jboss.seam.rest.validation.ValidationInterceptor</class>
</interceptors>
```

Then, enable validation of a particular method by decorating it with the `@ValidateRequest` annotation.

```java
@PUT
@ValidateRequest
public void updateTask(Task incommingTask)
{
...
}
```

Now, the HTTP request's entity body (the `incomingTask` parameter) will be validated prior to invoking the method.

58.1.1. Validating entity body

By default, the entity parameter (the parameter with no annotations that represent the body of the HTTP request) is validated. If the object is valid, the web service method is executed. Otherwise, a `ValidationException` exception is thrown.

The `ValidationException` exception is a simple carrier of constraint violations found by the Bean Validation provider. The exception can be handled by an `ExceptionMapper` or Solder exception handler.

Seam REST comes with a built-in `ValidationException` handler, which is registered by default. The exception handler converts the `ValidationException` to an HTTP response with the 400 (Bad request) status code. Furthermore, it sends messages relevant to the violated constraints within the message body of the HTTP response.

Example 58.1. HTTP response

```
HTTP/1.1 400 Bad Request
Content-Type: application/xml
Content-Length: 129

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error>
  <messages>
    <message>Name length must be between 1 and 100.</message>
  </messages>
</error>
```

58.1.2. Validating resource fields

Besides the message body, the JAX-RS specification allows various parts of the HTTP request to be injected into the JAX-RS resource or passed as method parameters. These parameters are usually HTTP form parameters, query parameters, path parameters, headers, etc.

Example 58.2. JAX-RS resource

```
public class PersonResource
{
    @QueryParam("search")
    @Size(min = 1, max = 30)
    private String query;
    @QueryParam("start")
    @DefaultValue("0")
    @Min(0)
    private int start;
    @QueryParam("limit")
    @DefaultValue("20")
    @Min(0) @Max(50)
    private int limit;
...
```

If a method of a resource is annotated with an `@ValidateRequest` annotation, the fields of a resource are validated by default.

> **Important**
>
> Since the JAX-RS injection occurs only at resource creation time, do not use the JAX-RS field injection for other than `@RequestScoped` resources.

58.1.3. Validating other method parameters

The JAX-RS specification allows path parameters, query parameters, matrix parameters, cookie parameters and headers to be passed as parameters of a resource method.

Example 58.3. JAX-RS method parameters

```
@GET
public List<Person>search(@QueryParam("search") String query,
    @QueryParam("start") @DefaultValue("0") int start,
    @QueryParam("limit") @DefaultValue("20") int limit)
```

> **Note**
>
> Currently, Seam REST validates only JavaBean parameters (as opposed to primitive types, Strings and so on). Therefore, to validate these types of parameters, either use resource field validation described in *Section 58.1.2, "Validating resource fields"* or read further and use parameter objects.

In order to prevent an oversized method signature when the number of parameters is too large, JAX-RS implementations provide implementations of the *Parameter Object pattern* [http://sourcemaking.com/refactoring/introduce-parameter-object] . These objects aggregate multiple parameters into a single object, for example *RESTEasy Form Object* [http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html/_Form.html] or *Apache CXF Parameter Bean* [http://cxf.apache.org/docs/jax-rs.html#JAX-RS-Parameterbeans] . These parameters can be validated by Seam REST. To trigger the validation, annotate the parameter with a `javax.validation.Valid` annotation.

Example 58.4. RESTEasy parameter object

```
public class MyForm {
  @FormParam("stuff")
  @Size(min = 1, max = 30)
  private int stuff;

  @HeaderParam("myHeader")
```

```
   private String header;

   @PathParam("foo")
   public void setFoo(String foo) {...}
}

@POST
@Path("/myservice")
@ValidateRequest
public void post(@Valid @Form MyForm form) {...}
```

58.2. Validation configuration

Table 58.1. `@ValidateRequest` annotation properties

`@ValidateRequest` attribute	Description	Default value
validateMessageBody	Enabling/disabling validation of message body parameters	true
validateResourceFields	Enabling/disabling validation of fields of a JAX-RS resource	true
groups	Validation groups to be used for validation	javax.validation.groups.Default

58.3. Using validation groups

In some cases, it is desired to have a specific group of constraints used for validation of web service parameters. These constraints are usually weaker than the default constraints of a domain model. Take partial updates as an example.

Consider the following example:

Example 58.5. Employee.java

```
public class Employee {
   @NotNull
   @Size(min = 2, max = 30)
   private String name;
   @NotNull
   @Email
   private String email;
   @NotNull
   private Department department;
```

```
    // getters and setters
}
```

The Employee resource in the example above is not allowed to have the null value specified in any of its fields. Thus, the entire representation of a resource (including the department and related object graph) must be sent to update the resource.

When using partial updates, only values of modified fields are required to be sent within the update request, while the non-null values of the received object are updated. Therefore, two groups of constraints are needed: group for partial updates (including `@Size` and `@Email`, excluding `@NotNull`) and the default group (@NotNull).

A validation group is a simple Java interface:

Example 58.6. `PartialUpdateGroup.java`

```
public interface PartialUpdateGroup {}
```

Example 58.7. Employee.java

```
@GroupSequence({ Default.class, PartialUpdateGroup.class })   3
public class Employee {
    @NotNull                                                  1
    @Size(min = 2, max = 30, groups = PartialUpdateGroup.class)  2
    private String name;
    @NotNull
    @Email(groups = PartialUpdateGroup.class)
    private String email;
    @NotNull
    private Department department;

    // getters and setters
}
```

1. The `@NotNull` constraint belongs to the default validation group.

2. The `@Size` constraint belongs to the partial update validation group.

3. The `@GroupsSequence` annotation indicates that both validation groups are used by default (for example, when persisting the entity).

Finally, the `ValidationInterceptor` is configured to validate the `PartialUpdateGroup` group only.

Example 58.8. `EmployeeResource.java`

```
@Path("/{id}")
  @PUT
  @Consumes("application/xml")
  @ValidateRequest(groups = PartialUpdateGroup.class)        1
  public void updateEmployee(Employee e, @PathParam("id") long id)
  {
    Employee employee = em.find(Employee.class, id);
    if (e.getName() != null)                                 2
    {
      employee.setName(e.getName());
    }
    if (e.getEmail() != null)
    {
      employee.setEmail(e.getEmail());
    }
  }
```

1. The partial update validation group is used for web service parameter validation.

2. Partial update — only the not-null fields of the transferred representation are used for update. The null fields are not updated.

Templating support

Seam REST allows to create HTTP responses based on the defined templates. Instead of being bound to a particular templating engine, Seam REST comes with a support for multiple templating engines and support for others can be plugged in.

59.1. Creating JAX-RS responses using templates

REST-based web services are often expected to return multiple representations of a resource. The templating support is useful for producing media formats such as XHTML and it can be also used instead of JAXB to produce domain-specific XML representations of a resource. Besides, almost any other representation of a resource can be described in a template.

To enable templating for a particular method, decorate the method with the `@ResponseTemplate` annotation. Path to a template file to be used for rendering is required.

Example 59.1. `@ResponseTemplate` in action

```
@ResponseTemplate("/freemarker/task.ftl")
public Task getTask(@PathParam("taskId") long taskId) {
...
}
```

The `@ResponseTemplate` annotation offers several other options. For example, it is possible for a method to offer multiple representations of a resource, each rendered with a different template. In the example below, the `produces` member of the `@ResponseTemplate` annotation is used to distinguish between produced media types.

Example 59.2. Multiple `@ResponseTemplate` S

```
@GET
@Produces( { "application/json", "application/categories+xml", "application/categories-short
+xml" })
@ResponseTemplate.List({
    @ResponseTemplate(value = "/freemarker/categories.ftl", produces = "application/categories
+xml"),
        @ResponseTemplate(value = "/freemarker/categories-short.ftl", produces = "application/
categories-short+xml")
})
public List<Category> getCategories()
```

Table 59.1. `@ResponseTemplate` **options**

Name	Required	Default value	Description
`value`	true	-	Path to the template (for example `/freemarker/categories.ftl`)
`produces`	false	*/*	Restriction of media type produced by the template (useful in situations when a method produces multiple media types, with different templates)
`responseName`	false	response	Name under which the object returned by the JAX-RS method is available in the template (for example, Hello ${response.name})

59.1.1. Accessing the model

There are several ways of accessing the domain data within a template.

Firstly, the object returned by the JAX-RS method is available under the "response" name by default. The object can be made available under a different name using the `responseName` member of the `@ResponseTemplate` annotation.

Example 59.3. hello.ftl

```
Hello ${response.name}
```

Secondly, every bean reachable via an EL expression is available within a template.

Example 59.4. Using EL names in a template

```
#foreach(${student} in ${university.students})
    <student>${student.name}</student>
#end
```

> **Note**
>
> Note that the syntax of the expression depends on the particular templating engine and mostly differs from the syntax of EL expressions. For example, `${university.students}` must be used instead of `#{university.students}` in a FreeMarker template.

Last but not least, the model can be populated programmatically. In order to do that, inject the `TemplatingModel` bean and put the desired objects into the underlying `data` map. In the following example, the list of professors is available under the "professors" name.

Example 59.5. Defining model programmatically

```java
@Inject
private TemplatingModel model;

@GET
@ResponseTemplate("/freemarker/university.ftl")
public University getUniversity()
{
  // load university and professors
  University university = ...
  List<Professor> professors = ...

  model.getData().put("professors", professors);
  return university;
}
```

> **Tip**
>
> When using JAXB-annotated classes as a return type for JAX-RS methods, you may see the following RESTEasy message: "Could not find JAXBContextFinder for media type: text/html" . This is caused by the built-in JAXB provider being too eager. You can prevent the built-in JAXB provider from being used for a method by adding the `@DoNotUseJAXBProvider` annotation on the method.

59.2. Built-in support for templating engines

Seam REST currently comes with built-in templating providers for FreeMarker and Apache Velocity.

59.2.1. FreeMarker

FreeMarker is one of the most popular templating engines. To enable Seam REST FreeMarker support, bundle the FreeMarker jar with the web application.

For more information on writing FreeMarker templates, refer to the *FreeMarker Manual* [http://freemarker.sourceforge.net/docs/index.html] .

59.2.2. Apache Velocity

Apache Velocity is another popular Java-based templating engine. Similarly to FreeMarker support, Velocity support is enabled automatically if Velocity libraries are detected on the classpath.

For more information on writing Velocity templates, refer to the *Apache Velocity User Guide* [http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html]

59.2.3. Pluggable support for templating engines

All that needs to be done to extend the set of supported templating engines is to implement the `TemplatingProvider` interface. Refer to *Javadoc* [http://docs.jboss.org/seam/3/rest/latest/api/org/jboss/seam/rest/templating/TemplatingProvider.html] for hints.

59.2.4. Selecting preferred templating engine

In certain deployment scenarios it is not possible to control the classpath completely and multiple template engines may be available at the same time. If that happens, Seam REST fails to operate with the following message:

Multiple TemplatingProviders found on classpath. Select the preferred one.

In such case, define the preferred templating engine in the XML configuration as demonstrated below to resolve the `TemplatingProvider` ambiguity.

Example 59.6. Preferred provider

```
<beans xmlns:rest="urn:java:org.jboss.seam.rest:org.jboss.seam.rest.exceptions">

 <rest:SeamRestConfiguration preferedTemplatingProvider="org.jboss.seam.rest.templating.freemarker.FreeMar
</beans>
```

Table 59.2. Built-in templating providers

Name	FQCN
FreeMarker	org.jboss.seam.rest.templating.freemarker.FreeMarkerProvider
Apache Velocity	org.jboss.seam.rest.templating.velocity.VelocityProvider

RESTEasy Client Framework Integration

The RESTEasy Client Framework is a framework for writing clients for REST-based web services. It reuses JAX-RS metadata for creating HTTP requests. For more information about the framework, refer to the *project documentation* [http://docs.jboss.org/resteasy/docs/2.0.0.GA/userguide/html/RESTEasy_Client_Framework.html] .

Integration with the RESTEasy Client Framework is optional in Seam REST and only available when RESTEasy is available on classpath.

> **Tip**
>
> Although RESTEasy is part of JBoss AS 7, not all of the required dependencies are exposed to the application classpath automatically. To enable RESTEasy Client Framework on JBoss AS 7, add the following line to `META-INF/MANIFEST.MF`
>
> Dependencies: org.apache.httpcomponents

60.1. Using RESTEasy Client Framework with Seam REST

Let us assume as an example that a remote server exposes a web service for providing task details to the client through the `TaskService` interface below.

Example 60.1. Sample JAX-RS annotated interface

```java
@Path("/task")
@Produces("application/xml")
public interface TaskService
{
    @GET
    @Path("/{id}")
    Task getTask(@PathParam("id")long id);
}
```

To access the remote web service, Seam REST builds and injects a client object of the web service.

Example 60.2. Injecting REST Client

```
@Inject @RestClient("http://example.com")
private TaskService taskService;

...

Task task = taskService.getTask(1);
```

The Seam REST module injects a proxied `TaskService` interface and the RESTEasy Client Framework converts every method invocation on the `TaskService` to an HTTP request and sends it over the wire to `http://example.com`. The HTTP response is unmarshalled automatically and the response object is returned by the method call.

URI definition supports EL expressions.

```
@Inject @RestClient("#{example.service.uri}")
```

60.2. Manual ClientRequest API

Besides proxying JAX-RS interfaces, the RESTEasy Client Framework provides the ClientRequest API for manual building of HTTP requests. For more information on the ClientRequest API, refer to the *project documentation* [http://docs.jboss.org/resteasy/ docs/2.0.0.GA/userguide/html/RESTEasy_Client_Framework.html#ClientRequest] .

Example 60.3. Injecting ClientRequest

```
@Inject @RestClient("http://localhost:8080/test/ping")
private ClientRequest request;

...

request.accept(MediaType.TEXT_PLAIN_TYPE);
ClientResponse<String> response = request.get(String.class);
```

60.3. Client Executor Configuration

If not specified otherwise, every request is executed by the default Apache HTTP Client 4 configuration. Provide an alternative `ClientExecutor` implementation to change this.

Example 60.4. Custom Apache HTTP Client 4 configuration

```
@Produces
public ClientExecutor createExecutor()
{
   HttpParams params = new BasicHttpParams();
   ConnManagerParams.setMaxTotalConnections(params, 3);
   ConnManagerParams.setTimeout(params, 1000);

   SchemeRegistry schemeRegistry = new SchemeRegistry();
   schemeRegistry.register(new Scheme("http", PlainSocketFactory.getSocketFactory(), 80));

   ClientConnectionManager cm = new ThreadSafeClientConnManager(params, schemeRegistry);
   HttpClient httpClient = new DefaultHttpClient(cm, params);

   return new ApacheHttpClient4Executor(httpClient);
}
```

Seam REST Dependencies

61.1. Transitive Dependencies

The Seam REST module depends on the *Seam Solder* [http://seamframework.org/Seam3/Solder] module.

61.2. Optional dependencies

61.2.1. FreeMarker

FreeMarker can be used for rendering HTTP responses. For more information on using FreeMarker with Seam REST, refer to *Section 59.2.1, "FreeMarker"*

```
<dependency>
   <groupId>org.freemarker</groupId>
   <artifactId>freemarker</artifactId>
   <version>${freemarker.version}</version>
</dependency>
```

61.2.2. Apache Velocity

Apache Velocity can be used for rendering HTTP responses. For more information on using Velocity with Seam REST, refer to *Section 59.2.2, "Apache Velocity"*

```
<dependency>
   <groupId>org.apache.velocity</groupId>
   <artifactId>velocity</artifactId>
   <version>${velocity.version}</version>
</dependency>
<dependency>
   <groupId>org.apache.velocity</groupId>
   <artifactId>velocity-tools</artifactId>
   <version>${velocity.tools.version}</version>
</dependency>
```

61.2.3. RESTEasy

RESTEasy Client Framework can be used for building clients of RESTful web services. For more information on using RESTEasy Client Framework, refer to *Chapter 60, RESTEasy Client Framework Integration*

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jaxrs</artifactId>
    <version>${resteasy.version}</version>
</dependency>
```

> **Note**
>
> Note that RESTEasy is provided on JBoss Application Server 6 and 7 and thus you do not need to bundle it with the web application.

Part XI. Seam JCR

Seam JCR - Introduction

62.1. Introduction

The Seam JCR Module aims to simplify the integration points between JCR implementations and CDI applications.

The Seam JCR module is compatible with JCR 2.0 implementations. It strictly compiles against JCR 2.0. However, test cases are executed against both ModeShape and JackRabbit to ensure compatibility.

62.2. 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 JCR:

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

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

<dependency>
  <groupId>org.jboss.seam.jcr</groupId>
```

```
    <artifactId>seam-jcr</artifactId>
    <version>${seam.jcr.version}</version>
    <scope>runtime</scope>
</dependency>
```

In addition to your Seam JCR dependencies, you must also declare a dependency on a JCR Implementation.

Seam JCR - JBoss ModeShape Integration

63.1. ModeShape Integration Installation

In order to activate ModeShape support within your application, you need to include ModeShape on your classpath. At a minimum, the following maven dependencies must be satisfied.

```xml
<dependency>
    <groupId>org.modeshape</groupId>
    <artifactId>modeshape-jcr</artifactId>
    <version>${modeshape.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-core</artifactId>
    <version>${lucene.version}</version>
</dependency>
```

Tip

Substitute ${modeshape.version} for the ModeShape version you are running against. Currently, Seam JCR tests against 2.5.0.Final. In addition, Lucene is required to run ModeShape. Please consult the ModeShape getting stated guide for exact versions.

63.2. Usage

In order to use ModeShape's Repository and Session objects in your application, you must define an injection point using the JcrConfiguration annotation based on ModeShape's required configuration parameters. Please review the ModeShape getting started guide for further details.

```java
        @Inject    @JcrConfiguration(name="org.modeshape.jcr.URL",value="file:path/to/
modeshape.xml?repositoryName=MyRepo")
    Repository repository;
```

```
            @Inject    @JcrConfiguration(name="org.modeshape.jcr.URL",value="file:path/to/
modeshape.xml?repositoryName=MyRepo")
   Session session;
```

Seam JCR - JackRabbit Integration

64.1. JackRabbit Integration Installation

In order to activate JackRabbit support within your application, you need to include JackRabbit on your classpath. At a minimum, the following maven dependency must be satisfied.

```xml
<dependency>
    <groupId>org.apache.jackrabbit</groupId>
    <artifactId>jackrabbit-core</artifactId>
    <version>${jackrabbit.version}</version>
</dependency>
```

Tip

Substitute ${jackrabbit.version} for the JackRabbit version you are running against. Currently, Seam JCR tests against 2.2.4. Please review JackRabbit documentation to determine any additional dependencies.

64.2. Usage

In order to use JackRabbit's Repository and Session objects in your application, you must define an injection point using the JcrConfiguration annotation based on JackRabbit's required configuration parameters.

```java
@Inject @JcrConfiguration(name="org.apache.jackrabbit.repository.home",value="target")
Repository repository;

@Inject @JcrConfiguration(name="org.apache.jackrabbit.repository.home",value="target")
Session session;
```

Seam JCR - Event Mapping

Seam JCR provides functionality to fire CDI Events based on events found in JCR. The rules of how events are fired are based around the underlying implementation.

> **Tip**
>
> You will not have an active JCR Session during the event firing, a new one will need to be created.

> **Tip**
>
> Some JCR implementations, like Modeshape fires events on a separate thread, so probably you will get errors if your observer method is declared on a @RequestScoped object, for example.

To observe an event, use the @Observes and the additional qualifiers on seam-jcr-api module (Check package org.jboss.seam.jcr.annotations.events). If you need to watch any JCR event, then avoid using any Qualifier at all.

```java
import javax.jcr.observation.Event;

public void observeAdded(@Observes @NodeAdded Event evt) {
    // Called when a node is added
}
public void observeAll(@Observes javax.jcr.observation.Event evt) {
    // Called when any node event occurs
}
```

Seam JCR - Object Content Mapping

66.1. What is Object Content Mapping?

Object Content Mapping is a design paradigm, in the same light as ORM (Object Relational Mapping) frameworks such as JPA or Hibernate, where statically typed objects are bound to a storage mechanism, in this case a JCR store. Seam JCR OCM is provided as annotations only on top of entities that are discovered during the CDI Phase ProcessAnnotatedType. In addition, Seam JCR's OCM implementation provides ServiceHandlers for working with entities over JCR.

66.2. Mapping and Conversion Capabilities

The mapping API is very simple and designed to be clean. In order to define an entity, you simply need to use the annotation `org.jboss.seam.jcr.annotations.ocm.JcrNode` to define that this is an entity to map. All fields by default will be mapped to their field names. You can override this behavior by using the annotation `org.jboss.seam.jcr.annotations.ocm.JcrProperty` which will map the property to a different property name. The `JcrProperty` annotation can be placed on both field and getter method. You can define a special property `uuid` of type String that will represent the identifier for the node. This is a sample node mapping:

```java
    @JcrNode("nt:unstructured")
public class BasicNode implements java.io.Serializable {
  @JcrProperty("myvalue")
  private String value;
  private String uuid;
  private String lordy;
  public String getValue() {
    return value;
  }
  public void setValue(String value) {
    this.value = value;
  }
  public String getUuid() {
    return uuid;
  }
  public void setUuid(String uuid) {
    this.uuid = uuid;
  }
  @JcrProperty("notaproperty")
  public String getLordy() {
    return lordy;
  }
```

```
    public void setLordy(String lordy) {
        this.lordy = lordy;
    }
}
```

The simplest way to convert entities is to use CDI Events. There are two event objects that can be fired to support parsing, `org.jboss.seam.jcr.ocm.ConvertToNode` and `org.jboss.seam.jcr.ocm.ConvertToObject`. By passing in a node and a pre-constructed object you can convert the full node to object or object to node depending on your need. Here is a sample parsing (from our test cases):

```
    @Inject Event<ConvertToObject< objectEvent;
    @Inject Event<ConvertToNode< nodeEvent;
    ....

    Node root = session.getRootNode();
    Node ocmnode1 = root.addNode("ocmnode1","nt:unstructured");
    BasicNode bn = new BasicNode();
    bn.setValue("Hello, World!");
    bn.setLordy("this was saved.");
    nodeEvent.fire(new ConvertToNode(bn,ocmnode1));

    Node hello2 = root.getNode("ocmnode1");
    BasicNode bn2 = new BasicNode();
    objectEvent.fire(new ConvertToObject(hello2,bn2));
```

66.3. JCR Data Access Objects

If you have ever worked with entities, the term DAO should be very familiar to you. Seam JCR OCM supports DAOs in a highly automated fashion. Using annotations and interfaces only, you can automate querying, finds and saving entities into their mapped node types. There are four annotations to support DAOs:

1. `org.jboss.seam.jcr.annotations.ocm.JcrDao` Defines this interface as a DAO interface. The ServiceHandler will be used to process these methods. This annotation should be placed at the interface level.

2. `org.jboss.seam.jcr.annotations.ocm.JcrFind` Defines this method as a find method, loading by identifier. The method should take a single String parameter and return a mapped node type.

3. `org.jboss.seam.jcr.annotations.ocm.JcrQuery` Defines this method as returning a `java.util.List` of mapped entities that can be mapped using the query results. Has properties defining the type to return, query to use, and the query language.

4. `org.jboss.seam.jcr.annotations.ocm.JcrParam` JcrParams are placed on the parameter arguments to a method annotated JcrQuery. Each argument should be a Value object and map based on bind parameters in the query.

Here is a sample definition of an interface, describing the objects that can be used:

```java
import static org.jboss.seam.jcr.ConfigParams.MODESHAPE_URL;

import java.util.List;

import org.jboss.seam.jcr.annotations.JcrConfiguration;
import org.jboss.seam.jcr.annotations.ocm.JcrDao;
import org.jboss.seam.jcr.annotations.ocm.JcrFind;
import org.jboss.seam.jcr.annotations.ocm.JcrQuery;
import org.jboss.seam.jcr.annotations.ocm.JcrSave;
import org.jboss.seam.jcr.test.ocm.BasicNode;

@JcrDao(
    @JcrConfiguration(name = MODESHAPE_URL,
        value = "file:target/test-classes/modeshape.xml?repositoryName=CarRepo")
)
public interface BasicNodeDAO {
    @JcrFind
    public BasicNode findBasicNode(String uuid);
                @JcrQuery(query="select    *    from    [nt:unstructured]",language="JCR-SQL2",resultClass=BasicNode.class)
    public List<BasicNode> findAllNodes();
    @JcrSave
    public String save(String path, BasicNode basicNode);
}
```

In this case, we are telling the JcrDao BasicNodeDAO to use the JCR Session based on the annotated JcrConfiguration noted. Since BasicNode is mapped to nt:unstructured, we can map any nt:unstructured to it by calling findAllNodes. We can save a basic node to a given path as well as find based on uuid. The best part is that there is no implementation necessary on your side. You can use this interface as is.

```
    @Inject
  BasicNodeDAO basicDAO;

  ....

    BasicNode bn = new BasicNode();
    bn.setValue("this is my node.");
    String uuid = basicDAO.save("/anypathone",bn);
    System.out.println("The UUID is: "+uuid);

    BasicNode bn2 = basicDAO.findBasicNode(uuid);
            System.out.printf("The  original  node  was  %s  and  the  new  node  is
\n",bn.getValue(), bn2.getValue());

    List<BasicNode> nodes = basicDAO.findAllNodes();
    System.out.println(nodes);
```

Part XII. Seam JMS

Introduction

Seam extends the CDI programming model into the messaging world by allowing you to inject JMS resources into your beans. Furthermore, Seam bridges the CDI event bus over JMS; this gives you the benefits of CDI-style type-safety for inter-application communication.

67.1. Mission statement

The JMS module for Seam 3 is to provide injection of JMS resources and the necessary scaffolding for a bidirectional propagation of CDI event over JMS.

67.2. Seam 3 JMS Module Overview

The general goals can be divided into two categories: injection of JMS resources and bridging of events:

JMS Resource Injection

- ConnectionFactory

- Connection

- Session

- Topics & Queues

- Message Producer

- Message Consumer

Event Bridge

- Inbound: Routes CDI events to JMS destinations

- Outbound: Fires CDI events based on the reception of JMS messages

Installation

Seam JMS can be used by including a few libraries in your application's library folder:

- seam-jms-api.jar

- seam-jms.jar

- solder-api.jar

- solder-impl.jar

- solder-logging.jar

If you are using *Maven* [http://maven.apache.org/] as your build tool use the following dependency, which will bring in both API and Implementation for Seam JMS:

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

> **Tip**
>
> Define or replace the property ${seam.jms.version} with a valid version of Seam JMS.

The runtime of Seam JMS is defined in two sections. The first section is related to creating observers, which happens within the Seam JMS CDI Extension. Observers need to be defined prior to starting up the container, and cannot be created once the application is running. This part happens automatically. The second section is related to creating listeners. This is managed in the component org.jboss.seam.jms.bridge.RouteBuilder.

> **Tip**
>
> In order to start any listeners, you may need to inject an instance of the RouteBuilder in to your class.
>
> If you are running within a Servlet Container, and include the Solder, RouteBuilder will automatically start up.
>
> The default implementation expects to find a ConnectionFactory at the JNDI location `/ConnectionFactory`. This can be changed by using Solder Config by

using a snippet similar to the one below in seam-beans.xml. This will change the JNDI location Seam JMS looks to `jms/ConnectionFactory`

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:java:ee"
    xmlns:jmsi="urn:java:org.jboss.seam.jms.inject"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://docs.jboss.org/cdi/beans_1_0.xsd">

  <jmsi:JmsConnectionFactoryProducer>
    <s:modifies />
              <jmsi:connectionFactoryJNDILocation>jms/ConnectionFactory</jmsi:connectionFactoryJNDILocation>
  </jmsi:JmsConnectionFactoryProducer>
  </beans>
```

Resource Injection

In this chapter we'll look at how to inject some of the common JMS resource types.

69.1. JMS Resource Injection

The following JMS resources are available for injection:

- javax.jms.Connection

- javax.jms.Session

Destination-based resources:

- javax.jms.Topic

- javax.jms.Queue

69.1.1. Destination Based Injection

The qualifier `@JmsDestination` is available to decorate JNDI oriented objects. This includes instances of `javax.jms.Destination` as well as MessageConsumers and MessageProducers.

```
@Inject @JmsDestination(jndiName="jms/MyTopic") Topic t;
@Inject @JmsDestination(jndiName="jms/MyQueue") Queue q;
```

69.1.2. Resource Configuration

You can use the `@JmsSession` annotation when injecting `javax.jms.Session` to specify transacted and acknowledgement mode:

```
@Inject @JmsSession(transacted=false, acknowledgementMode=Session.CLIENT_ACKNOWLEDGE) Session s;
```

69.2. Module Extensions

The Seam JMS module has certain points of extension, where the application developer can customize the behavior to match their needs. This is done by extending any of three base classes:

- org.jboss.seam.jms.inject.JmsConnectionFactoryProducer

- org.jboss.seam.jms.inject.JmsConnectionProducer

- org.jboss.seam.jms.inject.JmsSessionProducer

This can be done using CDI specializations and extending the base class to change the produced object. This allows the application developer to customize the produced behavior. For example, the base implementation assumes a Java EE container, however an extension to the JmsConnectionFactoryProducer could bootstrap a JMS container for you or change the default JNDI location of the `ConnectionFactory`

Each producer in these classes generates an instance based on The `@JmsDefault` annotation. This object is used within other places of the API, so you can control the Session generated that is injected into a MessageManager this way.

Messaging API

The Seam JMS Messaging API is a higher level abstraction of the JMS API to provide a number of convenient methods for creating consumers, producers, etc.

70.1. QueueBuilder and TopicBuilder

The `QueueBuilder` and `TopicBuilder` interfaces are meant to ease the integration of JMS while still sticking close to the base APIs. Within the single class you can work with both listeners and send messages. References to these classes can be injected or retrieved from a MessageManager instance (seen below).

Some example usages.

```
@RequestScoped
@Named
public class FormBean {
 private String formData;
 @Inject QueueBuilder queueBuilder;
 @Inject TopicBuilder topicBuilder;

 public void sendFormDataToQueue() {
    queueBuilder.destination("jms/SomeQueue").sendString(formData);
 }

 public void sendFormDataToTopic() {
    topicBuilder.destination("jms/SomeTopic").sendString(formData);
 }
}
```

70.2. Message Manager

The `MessageManager` interface (`org.jboss.seam.jms.MessageManager`) is the main consolidated API for Seam JMS. It provides almost all of the background functionality for Seam JMS's features (Observer Interfaces, Routing API). The default implementation works against `javax.naming.Context` assuming running within the same local application server.

```
public interface MessageManager {
   public ObjectMessage createObjectMessage(Object object);
   public TextMessage createTextMessage(String string);
   public MapMessage createMapMessage(Map<Object,Object> map);
   public BytesMessage createBytesMessage(byte[] bytes);
```

```
    public void sendMessage(Message message, String... destinations);
    public void sendObjectToDestinations(Object object, String... destinations);
    public void sendTextToDestinations(String string, String... destinations);
    public void sendMapToDestinations(Map map, String... destinations);
    public void sendBytesToDestinations(byte[] bytes, String... destinations);
    public void sendMessage(Message message, Destination... destinations);
    public void sendObjectToDestinations(Object object, Destination... destinations);
    public void sendTextToDestinations(String string, Destination... destinations);
    public void sendMapToDestinations(Map map, Destination... destinations);
    public void sendBytesToDestinations(byte[] bytes, Destination... destinations);
    public Session getSession();
    public MessageProducer createMessageProducer(String destination);
    public TopicPublisher createTopicPublisher(String destination);
    public QueueSender createQueueSender(String destination);
    public MessageConsumer createMessageConsumer(String destination, MessageListener... listeners);
    public MessageConsumer createMessageConsumer(Destination destination, MessageListener... listeners);
    public TopicSubscriber createTopicSubscriber(String destination, MessageListener... listeners);
   public QueueReceiver createQueueReceiver(String destination, MessageListener... listeners);
    public TopicBuilder createTopicBuilder();
    public QueueBuilder createQueueBuilder();
}
```

The interface above defines a full set of capabilities for creating and sending messages. In addition, we expose methods for creating producers (and A destination specific publisher and sender) as well as consumers (and A destination specific subscriber and receiver). In addition, if injected within a session scoped object, or similar, you can define a durable subscriber and unsubscriber for that subscriber. Below is an example.

The durable subscriber pattern works very well for session based message management. If you want to define a durable subscriber per user session, this is the easiest way to do it.

```
@SessionScoped
        public class UserSession {
            @Inject MessageManager messageManager;
            @Inject MySessionJMSListener listener;
            private String clientId;
            @PostConstruct
            public void registerListener() {
                clientId = UUID.randomUUID().toString();
                messageManager.createDurableSubscriber("jms/UserTopic",clientId,listener);
            }
            @PreDestroy
            public void shutdownListener() {
```

```
                    messageManager.unsubscribe(clientId);
                }
            }
```

70.3. Durable Messaging Capabilities

Seam JMS provides a Messaging API around the JMS Durable Topic Subscriber concept. In order to use it within your code, you need to inject a DurableMessageManager.

```
@Inject @Durable DurableMessageManager durableMsgManager;
```

This implementation of `MessageManager` provides additional methods to first login to the connection with a `ClientID`, additional methods to create subscribers and an unsubscribe that can be called to unsubscribe a listener.

```
public void login(String clientId);
public TopicSubscriber createDurableSubscriber(String topic, String id, MessageListener... listeners);
public TopicSubscriber createDurableSubscriber(Topic topic, String id, MessageListener... listeners);
public void unsubscribe(String id);
```

> **Tip**
>
> From a design pattern standpoint, it makes sense to create an `ApplicationScoped` object that all subscribers are created from, injecting a `DurableMessageManager` for use across the application, producing `SessionScoped` sessions for use by clients.

70.4. MessageListeners versus Message Driven Beans

One of the difficult choices we had to make was support for Message-Driven Beans. MDBs are a little complicated in CDI as they are not managed within the CDI life cycle. This makes integration with them a bit cumbersome. We wouldn't be able to work with a JMS Session in these cases, as an example. As a result, Seam JMS only supports defining instances of `javax.jms.MessageListener`. To support this, we have created a partial implementation - `org.jboss.seam.jms.AbstractMessageListener`. This special MessageListener is designed

for bridging the external context of a JMS Message into the application you are working with. We do this by tweaking classloaders.

The best way to work with MessageListeners is to simply instantiate your own based on our base implementation.

```
//where cl is the class loader, and beanManager is the BeanManager
MessageListener ml = new SessionListener(beanManager,cl,this);
messageManager.createTopicSubscriber("/jms/myTopic", ml);
```

Or you may define your own subclass that makes specific invocations to the parent. Here is an example of that:

```
@SessionScoped
        public class SessionListener extends AbstractMessageListener {
          private MySessionObject mso;
          public SessionListener(BeanManager beanManager, ClassLoader classLoader,
            MySessionObject mso){
            super(beanManager,classLoader);
            this.mso = mso;
          }

          @Override
          protected void handlMessage(Message msg) throws JMSException {
            //your business logic goes here
          }
        }
```

Bridging the Gap

This chapter is designed to detail how to configure the CDI to JMS event bridge. Routing has two sides, sending of events to JMS destinations and translating received messages from JMS destinations back into CDI events. The sections of this chapter describe how to achieve both.

71.1. Event Routing

Simply sending or receiving a message over JMS involves a few players: Connection, Session, Destination, and the message itself. Surely you can inject all required resources and perform the routing yourself but that takes away from the whole reason you're using a tool in the first place!

71.1.1. Routes

Routing CDI events to and from JMS can be configured by defining a `Route`. As you would normally create an observer method for an event you can define a route to control which events get forwarded to what destination. Or conversely, what message types sent to which destinations generate CDI events.

```java
public interface Route {
  public <D extends Destination> Route connectTo(Class<D> d, D destination);
  public Route addQualifiers(Annotation... qualifiers);

  ...
}
```

Routes allows for simple mapping of event types, complete with qualifiers, to a set of destinations. They can be configured by adding qualifiers and providing destinations they should interact with and are created from a `RouteManager`. Here's a simple route that forwards CDI events on to a queue:

```java
@EventRouting public Route registerMyRoute(RouteManager routeManager)
{
  Queue myQueue = lookupQueue("/jms/MyQueue");
  return routeManager.createRoute(RouteType.EGRESS, MyEvent.class).connectTo(Queue.class, myQueue);
}
```

A `RouteManager` is a factory object for creating new Routes. An instance of it is injected into every `@EventRouting` method. Classes with methods that are decorated with `EventRouting` must meet a few criteria items:

- A default, no arg constructor.

- Be a non bean (no dependencies on injection)

- Return either `Route` instances or `Collection<Route>` instances.

These requirements exist because of when the generation of `Route`s must happen. There are no CDI beans active within the context. A class identified for routing will automatically be veto'd from the context.

Routes are registered by returning them from a non-bean method annotated with `@EventRouting`:

```
@EventRouting public Route myConfig()
{
    return bridge.createRoute(RouteType.INGRESS, MyEvent.class).addDestinationJndiName("/
jms/MyTopic");
}
```

71.2. Routing CDI Events to JMS

Forwarding CDI events to JMS is configured by creating an egress route. Let's say you wanted to forward all `MyEvent` events with `@Bridged` qualifier to the queue `jms/EventQueue`. Simple, register a route:

```
AnnotationLiteral<Bridged> BRIDGED = new AnnotationLiteral<Bridged>() {};
@EventRouting public Route registerMyEventRoute(RouteManager routeManager)
{
    return routeManager.createRoute(RouteType.EGRESS, MyEvent.class).addQualifiers(BRIDGED).addDestinationJndiName("/
jms/EventQueue");
}
```

71.2.1. Usage

With your routing defined you can simply fire events that match the route's payload type and qualifiers and these events will be forwarded over JMS as object messages. A special note,

we have added the qualifier `@Routing(RouteType.EGRESS)`. This is necessary to avoid circular routings.

```
@Inject @Bridged @Routing(RouteType.EGRESS) Event<MyEvent> event;
...
event.fire(myEvent);
```

71.3. CDI Events from JMS Messages

Similar to egress routes, ingress routes are defined the same way. In this case, they listen for messages on the specified destination(s) and fire events. All of the data will be type safe, assuming you have defined your routes correctly.

Similar to the above example, this creates ingress routes from the Queue jms/EventQueue and fires events based on the MyEvent objects that are carried over the wire.

```
AnnotationLiteral<Bridged> BRIDGED = new AnnotationLiteral<Bridged>() {};
@EventRouting public Route registerMyEventRoute(RouteManager routeManager)
{
   return routeManager.createRoute(RouteType.INGRESS, MyEvent.class).addQualifiers(BRIDGED).addDestinationJndiName("/jms/EventQueue");
}
```

71.3.1. Usage

Once you define an ingress route, you handle it using an observer method. We use the same payload type and qualifiers, however we need to add the same qualifier, but for ingress `@Routing(RouteType.INGRESS)`

```
public void handleInboundMyEvent(@Observes @Routing(RouteType.INGRESS) MyEvent e) {
 ....
}
```

Annotation Routing APIs

This chapter is meant to describe the behavior of mapping interfaces, where event mapping to data flowing through JMS Queues and Topics are handled via events. These APIs are an alternate way to define routes as mentioned earlier in the document.

72.1. Observer Method Interfaces

Observer Method Interfaces are simple Plain Old Java Interfaces (POJIs) that define either a route. These interfaces exist within your code and are read at deployment time. This is a sample interface:

```java
public interface MappingInterface {
    @Inbound
    public void routeStringsFromTopic(@Observes String s, @JmsDestination(jndiName="jms/MyTopic") Topic t);

    @Outbound
    public void routeLongsToQueue(@Observes Long l, @JmsDestination(jndiName="jms/MyQueue") Queue q);

public void bidirectionRouteDoublesToQueue(@Observes Double d, @JmsDestination(jndiName="jms/DblQueue") Queue q);
}
```

This interface defines three routes. The first one being an ingress route - messages coming in to the topic jms/MyTopic will be fired as events with the type String. We indicate this by using the @Inbound annotation or @Routing(RouteType.INGRESS). The second being an egress route - events fired of type Long will be turned into ObjectMessages and using a MessageProducer sent to the queue jms/MyQueue. We indicate this by using the @Outbound annotation or @Routing(RouteType.EGRESS). The last is a bidirectional route, it defines messages that get fired in both directions. You can leave the method unannotated or use the @Routing(RouteType.BOTH) annotation.

The object being observed can have qualifiers. These qualifiers will be carried over in the fired event and follow the CDI rules for observer method selection. In all cases, the return type of the method is ignored.

The destinations can have any qualifier. In addition, there is basic support for @Resource on the method level to define the destination. This in general not 100% portable from the application developer perspective, we recommend heavy testing of the behavior on your application server.

In order to work with these routes, you raise events in CDI. In order to fire an event, first inject the `Event` object into your code with necessary annotations, for any egress route. For ingress routes, you need to define an observer method. Taking the third route as an example, here is how you would raise events to it

```
@Inject @Outbound Event<Double> doubleEvent
...
doubleEvent.fire(d);
```

and this is the appropriate observer method to handle the incoming messages.

```
public class MyEventObserverBean {
 public void sendMessage(@Observes @Inbound Double d) {
    System.out.println(d);
 }
}
```

Part XIII. Seam Validation

Introduction

The Seam Validation module aims at integrating *Hibernate Validator* [http://validator.hibernate.org/], the reference implementation for the Bean Validation API (*JSR 303* [http://jcp.org/en/jsr/detail?id=303]), with CDI (*JSR 299* [http://jcp.org/en/jsr/detail?id=299]).

This integration falls into two main areas:

- Enhanced dependency injection services for validators, validator factories and constraint validators

- Automatic validation of method parameters and return values based on Hibernate Validator's method validation feature

> **Note**
>
> The Seam Validation module is based on version 4.2 or later of Hibernate Validator. As of March 2011 Hibernate Validator 4.2 is still in the works and no final release exists yet.
>
> This means that - though unlikely - also changes to the API of the Seam Validation module might become necessary.
>
> The Seam Validation module is therefore released as a technology preview with the Seam 3 release train, with a final version following soon. Nevertheless you should give it a try already today and see what the Seam Validation module and especially the automatic method validation feature can do for you. Please refer to the *module home page* [http://seamframework.org/Seam3/ValidationModule] for any news on Seam Validation.

The remainder of this reference guide covers the following topics:

- *Installation* of Seam Validation

- *Dependency injection* services for Hibernate Validator

- Automatic *method validation*

Installation

This chapter describes the steps required to getting started with the Seam Validation Module.

74.1. Prerequisites

Not very much is needed in order to use the Seam Validation Module. Just be sure to run on JDK 5 or later, as the Bean Validation API and therefore this Seam module are heavily based on Java annotations.

74.2. Maven setup

The recommended way for setting up Seam Validation is using *Apache Maven* [http://maven.apache.org/]. The Seam Validation Module artifacts are deployed to the JBoss Maven repository. If not yet the case, therefore add this repository to your `settings.xml` file (typically in `~/.m2/settings.xml`) in order to download the dependencies from there:

Example 74.1. Setting up the JBoss Maven repository in settings.xml

```
...
<profiles>
  <profile>
    <repositories>
      <repository>
        <id>jboss-public</id>
        <url>http://repository.jboss.org/nexus/content/groups/public-jboss/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
    </repositories>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>jboss-public</activeProfile>
</activeProfiles>
...
```

General information on the JBoss Maven repository is available in the *JBoss community wiki* [http://community.jboss.org/wiki/MavenGettingStarted-Users], more information on Maven's `settings.xml` file can be found in the *settings reference* [???].

Having set up the repository you can add the Seam Validation Module as dependency to the `pom.xml` of your project. As most Seam modules the validation module is split into two parts, API and implementation. Generally you should be using only the types from the API within your application code. In order to avoid unintended imports from the implementation it is recommended to add the API as compile-time dependency, while the implementation should be added as runtime dependency only:

Example 74.2. Specifying the Seam Validation Module dependencies in pom.xml

```
...
<properties>
   <seam.validation.version>x.y.z</weld.version>
</properties>


...

<dependencies>
   ...
   <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>seam-validation-api</artifactId>
      <version>${seam.validation.version}</version>
      <scope>compile</scope>
   </dependency>
   <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>seam-validation</artifactId>
      <version>${seam.validation.version}</version>
      <scope>runtime</scope>
   </dependency>
   ...
</dependencies>
...
```

> **Note**
>
> Replace "x.y.z" in the properties block with the Seam Validation version you want to use.

74.3. Manual setup

In case you are not working with Maven or a comparable build management tool you can also add Seam Validation manually to you project.

Just download the latest distribution file from *SourceForge* [http://sourceforge.net/projects/jboss/files/Seam/Validation/], un-zip it and add seam-validation.jar api as well as all JARs contained in the lib folder of the distribution to the classpath of your project.

Dependency Injection

The Seam Validation module provides enhanced support for dependency injection services related to bean validation. This support falls into two areas:

- Retrieval of `javax.validation.ValidatorFactory` and `javax.validation.Validator` via dependency injection in non-Java EE environments

- Dependency injection for constraint validators

75.1. Retrieving of validator factory and validators via dependency injection

As the Bean Validation API is part of Java EE 6 there is an out-of-the-box support for retrieving validator factories and validators instances via dependency injection in any Java EE 6 container.

The Seam Validation module provides the same service for non-Java EE environments such as for instance stand-alone web containers. Just annotate any field of type `javax.validation.ValidatorFactory` with `@Inject` to have the default validator factory injected:

Example 75.1. Injection of default validator factory

```java
package com.mycompany;

import javax.inject.Inject;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class MyBean {

    @Inject
    private ValidatorFactory validatorFactory;

    public void doSomething() {

        Validator validator = validatorFactory.getValidator();
        //...
    }
}
```

> **Note**
>
> The injected factory is the default validator factory returned by the Bean Validation bootstrapping mechanism. This factory can customized with help of the configuration file `META-INF/validation.xml`. The Hibernate Validator Reference Guide *describes in detail* [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/validator-xmlconfiguration.html] the available configuration options.

It is also possible to directly inject a validator created by the default validator factory:

Example 75.2. Injection of a validator from the default validator factory

```java
package com.mycompany;

import java.util.Set;

import javax.inject.Inject;
import javax.validation.ConstraintViolation;
import javax.validation.Validator;

public class MyBean {

    @Inject
    private Validator validator;

    public void doSomething(Foo bar) {

        Set<ConstraintViolation<Foo>> constraintViolations = validator.validate(bar);
        //...
    }
}
```

75.2. Dependency injection for constraint validators

The Seam Validation module provides support for dependency injection within `javax.validation.ConstraintValidator` implementations. This is very useful if you need to access other CDI beans within you constraint validator such as business services etc. In order to make use of dependency injection within a constraint validator implementation it must be a valid bean type as described by the CDI specification, in particular it must be defined within a bean deployment archive.

> **Warning**
>
> Relying on dependency injection reduces portability of a validator implementation, i.e. it won't function properly without the Seam Validation module or a similar solution.

To make use of dependency injection in constraint validators you have to configure `org.jboss.seam.validation.InjectingConstraintValidatorFactory` as the constraint validator factory to be used by the bean validation provider. To do so create the file `META-INF/validation.xml` with the following contents:

Example 75.3. Configuration of InjectingConstraintValidatorFactory in META-INF/validation.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
          xmlns="http://jboss.org/xml/ns/javax/validation/configuration" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd">

  <constraint-validator-factory>
    org.jboss.seam.validation.InjectingConstraintValidatorFactory
  </constraint-validator-factory>

</validation-config>
```

Having configured the constraint validator factory you can inject arbitrary CDI beans into you validator implementations. Listing *Example 75.4, "Dependency injection within ConstraintValidator implementation"* shows a `ConstraintValidator` implementation for the `@Past` constraint which uses an injected time service instead of relying on the JVM's current time to determine whether a given date is in the past or not.

Example 75.4. Dependency injection within ConstraintValidator implementation

```java
package com.mycompany;

import java.util.Date;

import javax.inject.Inject;
```

```java
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import javax.validation.constraints.Past;

import com.mycompany.services.TimeService;

public class CustomPastValidator implements ConstraintValidator<Past, Date>
{

  @Inject
  private TimeService timeService;

  @Override
  public void initialize(Past constraintAnnotation)
  {
  }

  @Override
  public boolean isValid(Date value, ConstraintValidatorContext context)
  {

    if (value == null)
    {
      return true;
    }

    return value.before(timeService.getCurrentTime());
  }

}
```

> **Note**
>
> If you want to redefine the constraint validators for built-in constraints such as `@Past` these validator implementations have to be registered with a custom constraint mapping. More information can be found in the *Hibernate Validator Reference Guide* [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/validator-xmlconfiguration.html#d0e2024].

Method Validation

Hibernate Validator provides several advanced validation features and related functionality which go beyond what is defined by JSR 303 ("Bean Validation API"). One of these additional features is a facility for the validation of method parameters and return values. With that API a style of program design known as "Programming by Contract" can be implemented using the concepts defined by the Bean Validation API.

This means that any Bean Validation constraints can be used to describe

- any preconditions that must be met before a method may legally be invoked (by annotating method parameters with constraints) and

- any postconditions that are guaranteed after a method invocation returns (by annotating methods)

To give an example listing *Example 76.1, "Exemplary repository with constraint annotations"* shows a fictional repository class which retrieves customer objects for a given name. Constraint annotations are used here to express the following pre-/postconditions:

- The value for the name parameter may not be null and must be at least three characters long

- The method may never return null and each Customer object contained in the returned set is valid with respect to all constraints it hosts

Example 76.1. Exemplary repository with constraint annotations

```
@AutoValidating
public class CustomerRepository {

    @NotNull @Valid Set<Customer> findCustomersByName(@NotNull @Size(min=3) String name);

}
```

Hibernate Validator itself provides only an API for validating method parameters and return values, but it does not trigger this validation itself.

This is where Seam Validation comes into play. Seam Validation provides a so called business method interceptor which intercepts client invocations of a method and performs a validation of the method arguments before as well as a validation of the return value after the actual method invocation.

To control for which types such a validation shall be performed, Seam Validation provides an interceptor binding, `@AutoValidating`. If this annotation is declared on a given type an automatic validation of each invocation of any this type's methods will be performed.

If either during the parameter or the return value validation at least one constraint violation is detected (e.g. because `findCustomersByName()` from listing *Example 76.1, "Exemplary repository with constraint annotations"* was invoked with a String only two characters long), a `MethodConstraintViolationException` is thrown. That way it is ensured that all parameter constraints are fulfilled when the call flow comes to the method implementation (so it is not necessary to perform any parameter null checks manually for instance) and all return value constraints are fulfilled when the call flow returns to the caller of the method.

The exception thrown by Seam Validation (which would typically be written to a log file) gives a clear overview what went wrong during method invocation:

Example 76.2. Output of MethodConstraintViolationException

```
org.hibernate.validator.MethodConstraintViolationException: 1 constraint violation(s) occurred
 during method invocation.
Method:                                  public                                  java.lang.Set
 com.mycompany.service.CustomerRepository.findCustomersByName(java.lang.String)
Argument values: [B]
Constraint violations:
 (1) Kind: PARAMETER
    parameter index: 0
    message: size must be between 3 and 2147483647
      root bean: com.mycompany.service.org$jboss$weld$bean-flat-ManagedBean-class_com
$mycompany$service$$CustomerRepository_$$_WeldSubclass@3f72c47b
    property path: CustomerRepository#findCustomersByName(arg0)
                                                                              constraint:

 min=3, max=2147483647, payload=[], groups=[])
```

To make use of Seam Validation's validation interceptor it has to be registered in your component's beans.xml descriptor as shown in listing *Example 76.3, "Registering the validation interceptor in beans.xml"*:

Example 76.3. Registering the validation interceptor in beans.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
beans_1_0.xsd">

  <interceptors>
    <class>org.jboss.seam.validation.ValidationInterceptor</class>
```

```
    </interceptors>
</beans>
```

It is recommended that you consult the Hibernate Validator *reference guide* [http://docs.jboss.org/
hibernate/stable/validator/reference/en-US/html/] to learn more about the method validation
feature in general or for instance the rules that apply for constraining methods in inheritance
hierarchies in particular.

Part XIV. Seam Social

Introduction

The goal of Seam Social is to provide CDI components to access some of the social network services. It is focused on OAuth 1.0 and 2.0 services. At least Seam Social provides a way to get one's profile for a given service. There are services like Twitter for which Seam Social provides a lot of model object to manage timeline and friends in the service.

Seam social was created with genericity and polymorphism concepts in minds allowing users to deal with a collection of OAuth sessions, it also allows you to create your own extension to connect to an OAuth server.

This version of Seam Social provides support for the following OAuth Services with the following level

Table 8.

Service	Profile	Update	Timeline	Friends
Twitter	X	X	O	O
LinkedIn	X	O	O	O
Facebook	X	X	O	O

More services and level of support to come.

Getting Started

Provides CDI Beans and extensions to interact with major social network.

Provides:

- OAuth connectors to authentify with an OAuth providers

- Support for Authentication for Twitter, LinkedIn and Facebook only right now

- Status update for Facebook Twitter and LinkedIn

- Support for multi-account (multi-service and multi session for the same service)

Seam Social is independent of CDI implementation and fully portable between Java EE 6 and Servlet environments enhanced with CDI. It can be also used with CDI in JSE (desktop application). It is build on top of *scribe-java from fernandezpablo85* [https://github.com/fernandezpablo85/scribe-java]

For more information, see the *Seam Social project page* [http://seamframework.org/Seam3/Social].

77.1. Building

```
mvn -Pweld-ee-embedded-1.1 clean install
```

you need to be connected to internet to launch the tests. You can build without the tests like that :

```
mvn clean install -DskipTests=true
```

77.2. Usage big picture

The Web example app is quite simple and give a good idea of possibilities of Seam Social Framework.

Main steps to use Seam Social are :

- Declare an OAuth configuration

- Inject an OAuthService bean

- Request the Authorization URL for the service and get a request token

- Store the verifier in OAuthService bean and init access token

- Use the service

Should you need to fully understand each step, the complete OAuth lifecycle can be found *here* [https://dev.twitter.com/docs/auth/oauth] or *here* [https://developer.linkedin.com/documents/authentication]

77.3. Starting with OAuth configuration

To consume an OAuth service you need to declare an application on the service platform (i.e. for Twitter you can do, this on *https://dev.twitter.com/apps/new*). The declaration of an application contains at least :

- an API public key

- an API private/secret key

To use an OAuth service bean in Seam social you need to provide these configuration information in two ways :

- thru an OAuthConfigSettings bean

- by adding the @ConfigureOAuth annotation when injecting the OAuth service bean

77.3.1. Create an OAutConfigSettings bean thru Seam configuration (in bean.xml)

Right now, Seam Social provides only one convenient way to declare an OAuthConfigSettings bean. It can be done thru Seam configuration file (beans.xml). Here is an example of such a configuration :

```
<beans       xmlns="http://java.sun.com/xml/ns/javaee"       xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
   xmlns:s="urn:java:ee"
   xmlns:o="urn:java:org.jboss.seam.social.core">
   <o:OAuthServiceSettingsImpl>
     <s:modifies />
     <o:RelatedTo>Twitter</o:RelatedTo>
     <o:apiKey>FQzlQC49UhvbMZoxUIvHTQ</o:apiKey>
     <o:apiSecret>VQ5CZHG4qUoAkUUmckPn4iN4yyjBKcORTW0wnok4r1k
```

```
      </o:apiSecret>
      <o:callback>http://localhost:8080/social-web-client/callback.jsf
      </o:callback>
   </o:OAuthServiceSettingsImpl>
</beans>
```

Api Key and Api secret is provided by the service you want to consume (here Twitter). You can use the values above since they're coming from "Seam Social" Twitter application. Callback depends on your application : it's the URL that will collect OAuth verifier

77.3.2. Adding the @ConfigureOAuth annotation when injecting the OAuth service bean

You can simply add the @ConfigureOAuth annotation to the injection point. It can be done like that :

```
@Inject
@ConfigureOAuth(apiKey        =        "FQzlQC49UhvbMZoxUIvHTQ",        apiSecret        =
     "VQ5CZHG4qUoAkUUmckPn4iN4yyjBKcORTW0wnok4r1k",    callback="http://localhost:8080/
social-web-client/callback.jsf")
Twitter twitter;
```

With this notation the injected bean is configured with the given OAuth values.

77.4. Inject an OAuthService bean with one of the following ways :

Using the Interface of the service

```
@Named
@SessionScoped
public class mySessionBean implements Serializable {
   ...
   @Inject
   public Twitter twitter;
   ...
}
```

or using the generic OAuthService with a Qualifier

```
@Named
@SessionScoped
public class mySessionBean implements Serializable {
    ...
    @Inject
    @RelatedTo("Twitter")
    OAuthService service;
    ...
}
```

The two are equivalent but the second one give you a way to do polymorphic calls to the service. The OAuthService provides methods in relation to authentication.

77.5. Request the Authorization URL for the service and redirect the app to this url

If we go on with the same example, we can get this authorization URL with this call :

```
twitter.getAuthorizationUrl();
```

It will return the URL needed to initiate connection to the service.

77.6. Store the verifier in OAuthService bean and init access token

When we return from the service connection to the callback URL, we get a verifier that we need to store in the OAuthService and init the access token In JSF we do this like that

```
<f:metadata>
    <f:viewParam name="#{mySessionBean.twitter.verifierParamName}"
        value="#{mySessionBean.twitter.verifier}"
            required="true"
            requiredMessage="Error with Twitter. Retry later"/>
    <f:event type="preRenderView"
        listener="#{mySessionBean.twitter.initAccessToken()}"/>
</f:metadata>
```

77.7. After what we can send calls to the service

Getting the Twitter user profile

```
TwitterProfile user = twitter.getMyProfile();
String fullName = user.getFullName();
```

77.8. Testing

After building you can deploy the war generated in example/web-client/target in a Java EE 6 container implementing web profile (tested with JBoss 6 but should work in glassfish too)

Part XV. Seam Wicket

Introduction

The goal of Seam for Apache Wicket is to provide a fully integrated CDI programming model to the Apache Wicket web framework. Although Apache components (pages, panels, buttons, etc.) are created by direct construction using "new", and therefore are not themselves CDI contextual instances, with seam-wicket they can receive injections of scoped contextual instances via `@Inject`. In addition, conversation propagation is supported to allow a conversation scope to be tied to a wicket page and propagated across pages.

Installation

The `seam-wicket.jar` should be placed in the web application library folder. If you are using *Maven* [http://maven.apache.org/] as your build tool, you can add the following dependency to your `pom.xml` file:

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

> **Tip**
>
> Replace `${seam-wicket-version}` with the most recent or appropriate version of Seam for Apache Wicket.

As Wicket is normally used in a servlet (non-JavaEE) environment, you most likely will need to bootstrap the CDI container yourself. This is most easily accomplished using the Weld Servlet integration, described in the *Weld Reference Guide* [http://docs.jboss.org/weld/reference/latest/en-US/html/environments.html].

You must extend `org.jboss.seam.wicket.SeamApplication` rather than `org.apache.wicket.protocol.http.WebApplication`. In addition:

- if you override `newRequestCycleProcessor()` to return your own `IRequestCycleProcessor` subclass, you must instead override `getWebRequestCycleProcessorClass()` and return the class of your processor, and your processor must extend `SeamWebRequestCycleProcessor`.

- if you override `newRequestCycle` to return your own `RequestCycle` subclass, you must make that subclass extend `SeamRequestCycle`.

If you can't extend `SeamApplication`, for example if you use an alternate `Application` superclass for which you do not control the source, you can duplicate the three steps `SeamApplication` takes, i.e. return a `SeamWebRequestCycleProcessor` NonContextual instance in `newRequestCycleProcessor()`, return a `SeamRequestCycle` instance in `newRequestCycle()`, and add a `SeamComponentInstantiationListener` with `addComponentInstantiationListener()`.

Seam for Apache Wicket Features

Seam's integration with Wicket is focused on two tasks: conversation propagation through Wicket page metadata and contextual injection of Wicket components.

79.1. Injection

Any object that extends `org.apache.wicket.Component` or one of its subclasses is eligible for injection with CDI beans. This is accomplished by annotating fields of the component with the `@javax.inject.Inject` annotation:

```
public class MyPage extends WebPage {
  @Inject SomeDependency dependency;

  public MyPage()
  {
    depedency.doSomeWork();
  }
```

Note that since Wicket components must be serializable, any non-transient field of a Wicket component must be serializable. In the case of injected dependencies, the injected object itself will be serializable if the scope of the dependency is not `@Dependent`, because the object injected will be a serializable proxy, as required by the CDI specification. For injections of non-serializable `@Dependent` objects, the field should be marked transient and the injection should be looked up again in a component-specific attach() override, using the `BeanManager` API.

Upon startup, the CDI container will examine your component classes to ensure that the injections you use are resolvable and unambiguous, as per the CDI specification. If any injections fail this check, your application will fail to bootstrap.

The scopes available are similar to those in a JSF application, and are described by the CDI specification. The container, in a Java EE environment, or the Servlet listeners, in a Servlet environment, will set up the application, session, and request scopes. The conversation scope is set up by the `SeamWebRequestCycle` as outlined in the next two sections.

79.2. Conversation Control

Application conversation control is accomplished as per the CDI specification. By default, like JSF/CDI, each Wicket HTTP request (whether AJAX or not) has a *transient* conversation, which is destroyed at the end of the request. A conversation is marked *long-running* by injecting the `javax.enterprise.context.Conversation` bean and calling its `begin()` method.

```
public class MyPage extends WebPage {
```

```
@Inject Conversation conversation;

public MyPage()
{
  conversation.begin();
  //set up components here
}
```

Similarly, a conversation is ended with the `Conversation` bean's `end()` method.

79.3. Conversation Propagation

A transient conversation is created when the first Wicket `IRequestTarget` is set during a request. If the request target is an `IPageRequestTarget` for a page which has previously marked a conversation as non-transient, or if the *cid* parameter is present in the request, the specified conversation will be activated. If the conversation is missing (i.e. has timed out and been destroyed), `SeamRequestCycle.handleMissingConversation()` will be invoked. By default this does nothing, and your conversation will be new and transient. You can however override this, for example to throw a `PageExpiredException` or something similar. Upon the end of a response, `SeamRequestCycleProcessor` will store the *cid* of a long running conversation, if one exists, to the current page's metadata map, if there is a current page. The key for the *cid* in the metadata map is the singleton `SeamMetaData.CID`. Finally, upon `detach()`, the `SeamRequestCycle` will invalidate and deactive the conversation context.

Note that the above process indicates that after a conversation is marked long-running by a page, requests made back to that page (whether AJAX or not) will activate that conversation. It also means that new `Page` objects assigned as a `RequestTarget`, whether directly via `setResponsePage(somePageInstance)` or with `setResponsePage(SomePage.class, pageParameters)`, will have the conversation propagated to them. This can be avoided by:

1. ending the conversation before the call to setResponsePage,

2. using a BookmarkablePageLink rather than directly instantiating the response page, or

3. specifying an empty *cid* parameter in `PageParameters` when using `setResponsePage()`.

> **Note**
>
> The final case also provides a mechanism for switching conversations: if a *cid* is specified in `PageParameters`, it will be used by bookmarkable pages, rather than the current conversation.