Solder

Reference Guide

by Pete Muir, Stuart Douglas, Dan Allen, John Ament, Shane Bryzak,
Jason Porter, Lincoln Baxter III, Nicklas Karlsson, and Christian Kaltepoth

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. *Part I, "Extensions and Utilities for Developers"* details extensions and utilities which are likely to be of use to any developer using CDI; *Part II, "Utilities for Framework Authors"* describes utilities which are likely to be of use to developers writing libraries and frameworks that work with CDI; *Part III, "Configuration Extensions for Framework Authors"* 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.

1.1. 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:

```
<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!*

1.2. 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:

```
<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 6, Logging, redesigned* for more information about how logging is handled in Solder.

1.3. 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!

Part I. Extensions and Utilities for Developers

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*.

2.1. Preventing a class from being processed

2.1.1. @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 2.1. 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.

2.1.2. @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.

2.2. @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;

  ...

}
```

2.3. @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.

2.4. 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
```

2.5. @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.

```
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.

```
@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:

```
class FruitBowl {

   @Inject Expressions expressions;

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

4.1. @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`.

Injecting Resources and System Properties

5.1. 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;
```

5.1.1. 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`.

5.2. System Properties

Solder allows system properties to be easily injected using the `@System` qualifier. The following code snippet shows how you can inject system properties directly into your own bean:

```
import java.util.Properties;
import org.jboss.solder.core.System;
import javax.inject.Inject;

public class Foo {
    @Inject @System Properties properties;

    //..
}
```

Solder also exposes the system properties as a named bean called `sysProp`, allowing them to be referenced directly via EL (Expression Language), for example from a JSF page definition. Please refer to the `org.jboss.solder.system.SystemProperties` class in the Solder API documentation for a list of the available methods.

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!).

6.1. 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.

6.2. 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.

6.3. Typed loggers

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

```
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.

6.4. 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;
```

6.5. 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, annotate 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"));
```

6.6. 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!

6.6.1. 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>
```

```
    <!-- 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).

6.6.2. 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).

Part II. Utilities for Framework Authors

Annotation and AnnotatedType Utilities

Solder provides a number of utility classes that make working with annotations and `AnnotatedTypes` 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.

7.1. 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.

7.2. 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);
```

7.3. 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 `Stereotypes` 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 `Annotations` on `Stereotypes`, 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 `Stereotypes` or not. For both of these methods, a search is performed first directly on the element before searching in stereotypes.

7.4. 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();
```

7.5. 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:

```
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:

```
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.

Property Utilities

Solder provides a number of convenient features for querying 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 to work with all properties in the same way.

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

10.1. 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 10.1. 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`:'

```
class Person {

  PersonName personName;

  Address address;

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

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

}
```

You can create two properties:

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

10.2. Querying for properties

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

```
  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:

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

10.3. 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.

10.3.1. AnnotatedPropertyCriteria

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

```
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:

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

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

10.3.2. NamedPropertyCriteria

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

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

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

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

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

10.3.3. 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));
```

10.3.4. 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;
  }
}
```

10.4. 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`.

Part III. Configuration Extensions for Framework Authors

Unwrapping Producer Methods

Unwrapping producer methods allow you to create injectable objects that have "self-managed" lifecycles. An unwrapped injectable object is useful if you need a bean whose lifecycle does not exactly match one of the lifecycles 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:

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

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

}
```

And in the JSF module:

```
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.

13.1. 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>
```

13.2. 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:

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

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

```
@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:

```
class QueryHandler {

  @Inject EntityManager em;

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

```
}
```

Note

- The invocation handler is similar to an interceptor. It must have an `@AroundInvoke` method that returns an 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:

```
@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.

```
class UserListManager {
  @Inject
  UserQuery userQuery;

  List<User> users;

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

}
```

Part IV. XML Configuration

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.

15.1. 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:

```
class Report {
 String filename;

 @Inject
 Datasource datasource;

 //getters and setters
}
```

And the following support classes:

```
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.

15.2. 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

16.1. 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*.

16.2. 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.

16.3. 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.

16.4. 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/>
```

16.4.1. 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.

16.4.2. 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>
```

16.5. Configuring methods

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

```
class MethodBean {

   public int doStuff() {
      return 1;
   }

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

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

}
```

```
<?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.

16.6. 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)
  {
    ...
```

```
  }
}
```

16.7. 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;
}
```

```
<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.

16.8. 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:

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

This configures a qualifier annotation:

```
<my:SomeQualifier>
```

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

This configures an interceptor binding:

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

16.9. 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.

16.10. 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.

Part V. Exception Handling Framework

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.

17.1. 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

18.1. 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

18.1.1. 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.

18.1.2. 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.

18.2. 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.

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`.

18.3. 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.

18.3.1. @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.

18.3.2. @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 18.5.1, "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.

18.4. 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`?

18.5. 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.

18.5.1. 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).

18.5.2. 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:

```
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

18.6. APIs for exception information and flow control

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

- `CaughtException`
- `ExceptionStack`

18.6.1. 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`.

18.6.2. 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

19.1. Exception Modification

19.1.1. 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.

19.1.2. 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.

19.2. Filtering Stack Traces

19.2.1. 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

19.2.2. 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.

19.2.3. 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.

19.2.4. 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.

19.2.5. 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 19.1. Terminate

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

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

Example 19.2. 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 19.3. 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 19.4. 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

20.1. 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.

20.2. Default Handlers and Qualifiers

20.2.1. 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.

20.2.2. 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.

20.3. 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 `ServiceHandlers` simply do not make sense.

Note

If ServiceHandlers are implemented make sure to document if any of the methods are called from `CaughtException`, specifically `abort()`, `handled()` or `rethrow()`.

> These methods affect invocation of other handlers (or rethrowing the exception in the case of `rethrow()`).

20.4. 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 `@Handlers` 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 VI. Servlet API Integration

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

21.1. 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:

```
<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.

22.1. 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.

22.2. 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");
}
```

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

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

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

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

22.4. 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	A HTTP servlet response is initialized or destroyed
@Initialized	javax.servlet.http.HttpServletResponse	A HTTP servlet response is initialized
@Destroyed	javax.servlet.http.HttpServletResponse	A 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.

22.5. 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!

22.6. 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.

22.7. 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.

23.1. @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();
```

23.2. @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();
```

23.3. @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.

23.4. @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.

23.5. @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.

23.6. @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.

23.7. @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();
}
```

23.8. @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`.

23.9. @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()`.

23.10. @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();
```

23.11. @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`.

23.12. @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.

24.1. 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.

24.2. 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:

```
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 8, 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.

Loading web resources without ServletContext

Sometimes developers need to access web application resources from application code. Typically the `ServletContext` is used to load resources by calling `getResource()`. Unfortunately the `ServletContext` cannot be accessed in all situations. Especially CDI extensions can be problematic in this regard as they are executed during a stage in the application startup in which the `ServletContext` may not have been created yet.

Solder offers some help in this situation. The class `WebResourceLocator` provides a simple way to obtain resources from the web application. Under the covers this class uses the `WebResourceLocationProvider` SPI to retrieve the location of the resources.

The following example shows how to use the class:

```
WebResourceLocator locator = new WebResourceLocator();

InputStream stream = locator.getWebResource("/WEB-INF/web.xml");

if (stream != null) {

  // parse the input stream

}
```

As you can see using the `WebResourceLocator` is very easy. Just create an instance of the class and then use `getWebResource()` to retrieve an InputStream.

Warning

Please note that you should always prefer to use the standard Servlet API to load resources from the web application if possible. This Solder API is only intended to be used if it is not possible to use the `ServletContext` (like for example in CDI extensions).