Weld Extensions

Introduction

Weld Extensions 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 Weld Extensions is easy. If you are using Maven, then you can declare a dependency on Weld Extensions (`org.jboss.weld:weld-extensions:${weld.extensions.version}`, make sure you have the JBoss Maven repository enabled). Otherwise, add the jar to your compile time and runtime classpath.

Most of Weld Extensions has very few dependencies:

- `javax.enterprise:cdi-api`
- `org.slf4j:slf4j-api`
- `org.jboss.logging:jboss-logging-api`
- `javax.el:el-api`
- `javax.inject:javax.inject`
- `javax.transaction:jta`

Tip

The POM for Weld Extensions 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.weld</groupId>
   <artifactId>weld-extensions</artifactId>
   <version>${weld.extensions.version}</version>
   <type>pom</type>
   <scope>import</scope>
</dependency>
```

Some features of Weld Extensions 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

Part I. Extensions and Utilities for Developers

Enhancements to the CDI Programming Model

Weld Extensions 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 {
   ...
}
```

Note

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

2.1.2. @Requires

Annotating a class `@Requires` will cause the type to be ignored if the class dependencies can 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

Weld 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(EntityManager.class)
class EntityManagerProducer {

   @Produces
   EntityManager getEntityManager() {
      ...
   }

}
```

Note

The `ProcessAnnotatedType` container lifecycle event will be called for vetoed 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). Weld Extensions provides a built in qualifier, `@Client` for this purpose.

2.4. Named packages

Weld Extensions 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
```

Annotation Literals

Weld extensions provides a complete set of `AnnotationLiterals` for every annotation type defined by the CDI (JSR-299) and Injection (JSR-330) specification. These are located in the `org.jboss.weld.extensions.literal` package. Annotations without listitems provide a static `INSTANCE` listitem that should be used rather than creating a new instance every time.

Literals are provided for the following annotations from *Context and Dependency Injection*:

- `@Alternative`
- `@Any`
- `@ApplicationScoped`
- `@ConversationScoped`
- `@Decorator`
- `@Default`
- `@Delegate`
- `@Dependent`
- `@Disposes`
- `@Inject`
- `@Model`
- `@Named`
- `@New`
- `@Nonbinding`
- `@NormalScope`
- `@Observes`
- `@Produces`
- `@RequestScoped`
- `@SessionScoped`
- `@Specializes`
- `@Stereotype`
- `@Typed`

Literals are provided for the following annotations from *Weld Extensions*:

- `@Client`
- `@DefaultBean`
- `@Exact`

- `@Generic`
- `@GenericType`
- `@Mapper`
- `@MessageBundle`
- `@Requires`
- `@Resolver`
- `@Resource`
- `@Unwraps`
- `@Veto`

Evaluating Unified EL

Weld extensions 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 Weld Extensions 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}");
   }
}
```

Resource Loading

Weld Extensions 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. 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.weld.extensions.resourceLoader.ResourceLoader`.

Logging

Weld Extensions integrates JBoss Logging 3 as it's logging framework of choice. JBoss Logging 3 is a modern logging framework offering:

- Abstracts away from common logging backends and frameworks (such as JDK Logging, log4j and slf4j)
- Provides a innovative, typed logger (see below for examples)
- Full support for internationalization and localization
  - Developers can work with interfaces and annotations only
  - Translators can work with message bundles in properties files
- Build time tooling to generate typed loggers for production, and runtime generation of typed loggers for development
- Access to MDC and NDC (if underlying logger supports it)
- Loggers are serializable

Note

A number of the features of JBoss Logging 3 are still under development - at the moment only runtime generation of typed is supported, and these loggers only support the default message placed on the typed logger, and will not look up a localized message.

To use a typed logger, first create the logger definition:

```
@MessageLogger
interface TrainSpotterLog {

   // Define log call with message, using printf-style interpolation of parameters
   @LogMessage @Message("Spotted %s diesel trains")
   void dieselTrainsSpotted(int number);

}
```

You can then inject the typed logger with no further configuration:

```
   // Use the train spotter log, with the log category "trains"
   @Inject @Category("trains") TrainSpotterLog log;
```

and use it:

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

JBoss Logging will use the default locale unless overridden:

```
   // Use the train spotter log, with the log category "trains", and select the UK locale
   @Inject @Category("trains") @Locale("en_GB") TrainSpotterLog log;
```

You can also log exceptions:

```
@MessageLogger
interface TrainSpotterLog {

   // Define log call with message, using printf-style interpolation of parameters
   // The exception parameter will be logged as an exception
   @LogMessage @Message("Failed to spot train %s")
   void missedTrain(String trainNumber,@Cause Exception exception);

}
```

You can then log a message with an exception:

```
log.missedTrain("RH1", cause);
```

You can also inject a "plain old" Logger:

```
   @Inject Logger log;
```

Typed loggers also provide internationalization support, simply add the @MessageBundle annotation to the logger interface (not currently supported).

Sometimes you need to access the message directly (for example to localize an exception message). Weld Extensions let's you inject a typed message bundle. First, declare the message bundle:

```
@MessageBundle
interface TrainMessages {

   // Define a message using printf-style interpolation of parameters
   @Message("No trains spotted due to %s")
   String noTrainsSpotted(String cause);

}
```

Inject it:

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

And use it:

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

Part II. Utilities for Framework Authors

Annotation and AnnotatedType Utilities

Weld Extensions provides a number of utilility classes to make working with Annotations and AnnotatedTypes easier. This chapter will walk you each utility, and give you an idea of how to use it. For more detail, take a look at the javaodoc on each class.

7.1. Annotated Type Builder

Weld Extensions provides an `AnnotatedType` implementation that should be suitable for most portable extensions needs. The `AnnotatedType` is created from `AnnotatedTypeBuilder` as follows:

```
AnnotatedTypeBuilder builder = new AnnotatedTypeBuilder()
      .readFromType(baseType,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 it's members. When we have finished modifying the type, we call `create()` to spit out a new, immutable, `AnnotatedType`.

`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 any class member and the type:

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

         public void redefine(RedefinitionContext<A> ctx) {
            ctx.getAnnotationBuilder().remove(Unique.class);
         }

      }
      .create();
```

7.2. Annotation Instance Provider

Sometimes you may need an annotation instance for an annotation whose type is not known at development time. Weld extends 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);
```

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. Weld Extension's synthetic qualifers can be used to easily generate and track such qualifers.

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

Weld Extensions 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.

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

Obtaining a handle on the BeanManager

When developing a framework that builds on CDI, you may need to obtain the `BeanManager` for the application, 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 Weld Extensions. To lookup up a `BeanManager`, you can extend the `BeanManagerAware` class, and call `getBeanManager`:

```
class WicketIntegration extends BeanManagerAware {

   public WicketManager getWicketManager() {
      Bean<?> bean = getBeanManager.getBean(Instance.class);
      ...
   }

   ...

}
```

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

```
class ResourceServlet extends HttpServlet {

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

Bean Utilities

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

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

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

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

`Beans`
: A set of utilities for working with beans.

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

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

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

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

`ImmutableInjectionPoint`
: An immutable (and hence thread-safe) injection point.

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

`ImmutablePassivationCapableBean`
: An immutable (and hence thread-safe) bean, whose constructor will substitute specification defaults if `null` is passed for a particular attribute. An implementation of `ContextualLifecycle` may be registered to receive lifecycle callbacks. The bean implements `PassivationCapable`, and an id must be provided.

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

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

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

Properties

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

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

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.	

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. Weld Extensions 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.weld.extensions.util.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.class ||
             f.getType() == Long.class || f.getType() == Long.TYPE.class ||
             f.getType() == BigInteger.class;
   }

   boolean methodMatches(Method m) {
      return m.getReturnType() == Integer.class || m.getReturnType() == Integer.TYPE.class ||
             m.getReturnType() == Long.class || m.getReturnType() == Long.TYPE.class ||
             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(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(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.

Part III. Configuration Extensions for Framework Authors

Unwrapping Producer Methods

Unwrapping producer methods allow you to create injectable objects that have "self-managed"" lifecycles, and are particularly useful if you have need a bean whose lifecycle does not exactly match one of the lifecycle of one of the existing scopes. The lifecycle of the bean is are 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"). Every time a method is called on unwrapped object the invocation is forwarded to the result of calling the unwrapping producer method.

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;
   }
}
```

```
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 repreents) 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 Weld Extensions 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(type = FunctionMapper.class)
@Mapper
class FunctionMapperImpl extends FunctionMapper {

   @Override
   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 provides 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's as they need, however we do not want to have to declare each producers 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 Weld Extensions 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) {
      ...
   }
}
```

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 name();

}
```

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.

Important

All generic configuration annotations should have at least one member. Each configuration must specify a unique value for this member.

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 it's 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 enqueing messages. The client can then tweak the policy should they wish.

```
@Generic(ACMEQueue.class)
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:

```
@ServiceHandler(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 intercepter. It must have an `@AroundInvoke` method that returns and object and takes an `InvocationContext` as an argument.
- Do not call `InvocationContext.proceed()` as there is no method to proceed to.

- Injection is available into the handler class, however the handler is not a bean definition, so observer methods, producer fields and producer methods defined on the handler will not be registered.

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

```
@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();
   }

}
```