Java Injection (CanDI) Pattern Tutorial

June 15, 2009

Contents

1 Overview

The four main CanDI patterns share a common goal: improve code with a declarative injection style. When Java classes cleanly describe their dependencies, their lifecycles, and their exports, they are self-documenting. You can read the classes and understand how they will behave, i.e. you don't need to read the code side-by-side with XML configuration to understand the system's behavior.

The custom, typesafe binding annotations are key to CanDI's self-documentation, because injection points clearly describe the resources or services they expect with adjective-oriented annotations. Because the annotations are true Java classes, they are documented in JavaDoc and verified by the compiler. The small number of meaningful adjectives means they don't impose a significant coding burden, and are well worth the small extra development time.

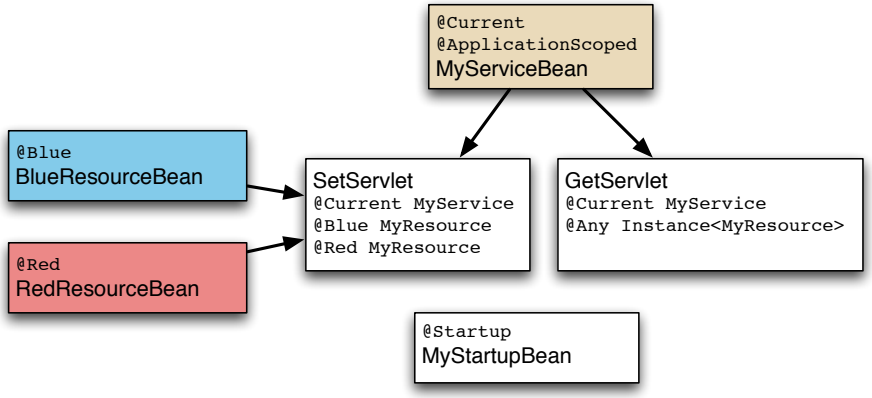
CanDI Application Patterns

PATTERN	DESCRIPTION
Service Pattern	Organize the application as a collection of services.

Resource Configuration Pattern	Bind and resources with declarative annotations and configure with XML.
Startup Pattern	Use @Startup beans to initialize application state.
Plugin/Extension Pattern	Discover plugin/extension classes for a service.

This tutorial describes four main CanDI design patterns: services, resources, startup and extensions. Services center an application's design by encapsulating management and data. Resources are the configurable interface between the user and the application. Startup initializes the application. And extensions allow sophisticated applications to tailor their behavior to the user's needs.

1.1 Tutorial Architecture



Since the purpose of the service pattern is encapsulating state and managment for multiple clients, the tutorial shows a single service used by multiple servlets and by PHP and JSP scripts. Services are typically singletons in the application and use `@Current` to mark the binding.

The resource pattern configures a driver class and properties in XML for an application resource. The resource tutorial uses `MyResource` as a general resource API, like `DataSource` or `EntityManager` , and application specific bindings `@Red` and `@Blue` . Because resource APIs are general, they need an application-specific description to document their purpose in the code. Binding annotations are simple, clear adjectives, and typically only a small number are needed. The driver classes like `BlueResourceBean` are typically selected and configured in an XML, like selecting and configuring a database.

Startup initialization is needed by most applications, and can use the CanDI startup pattern to document the startup classes and avoid unnecessary XML.

Because CanDI discovers beans through classpath scanning, you can create startup beans with just a `@Startup` annotation and a `@PostConstruct` method.

A plugin or extension capability can improve the flexibility of many applications, even if designed for internal use. The plugin pattern uses CanDI's discovery process for the plugin capability without requiring a new infrastructure. The tutorial reuses the `MyResource` API as a plugin API and grab all implementations using the CanDI `Instance` interface and the `@Any` annotation.

1.2 Java Injection API

The most important CanDI classes are just three annotations: `@Current` , `@BindingType` and `@ApplicationScoped` , because many applications will primarily use the service and resource patterns. By using these three annotations effectively, you can improve the readability and maintainability of your application's services and resources.

Service and Resource Pattern CanDI classes

ANNOTATION/CLASS	DESCRIPTION
@ApplicationScoped	scope annotation marking the service as a singleton
@BindingType	descriptive application bindings are marked with this meta-annotation
@Current	Default binding for unique beans (service pattern).

Applications which provide scripting access to services or resources will use the `@Named` annotation to provide a scripting name for the beans.

Scripting Support CanDI classes

ANNOTATION/CLASS	DESCRIPTION
Named	Scriping and JSP/JSF EL access to CanDI beans (service pattern)

The startup pattern uses two additional annotations, `@Startup` to mark the bean as needing creation on container start, and `@PostConstruct` marking a method to be initialized.

Startup Pattern CanDI classes

ANNOTATION/CLASS	DESCRIPTION
@Startup	Starts a bean when the container starts.
@PostConstruct	Calls an initialization method the bean is created.

A plugin or extension architecture uses two additional CanDI classes to easily find plugins discovered during CanDI's classpath scanning. `Instance<T>` provides an iterator over all the discovered and configured beans, and `@Any` selects all beans independent of their `@BindingType` .

Plugin/Extension Pattern CanDI classes

ANNOTATION/CLASS	DESCRIPTION
Instance<T>	Programmatic access to all implementations of an interface.
@All	Selects all matching beans for an interface.

2 Service Pattern

Because services are often unique in an application, the service interface is generally enough to uniquely identify the service. In CanDI, the `@Current` annotation injects a unique service to a client class. A declarative style applies to both the service declaration and the service use, by annotating the service scope as `@ApplicationScoped` , and annotating the client injection as `@Current` . By describing the function on the class itself, CanDI's annotations improve the readability and maintainability of service classes.

```
package example;

import javax.enterprise.inject.Current;
...

public class GetServlet extends HttpServlet {
  private @Current MyService _service;

  ...
}
```

Example: GetServlet.java

Users of the service will access it through an interface like `MyService` . The implementation will be a concrete class like `MyServiceBean` . The interface API in CanDI is a plain Java interface with no CanDI-specific annotations or references.

```
package example;

public interface MyService {
  public void setMessage(String message);

  public String getMessage();
}
```

Example: MyService.java

All the information relevant to the class deployment is on the class itself, because the service implementation is discovered through CanDI's classpath scanning. In other words, The service's deployment is self-documenting. Since services are generally singletons, they will typically have the `@ApplicationScoped` annotation. Other annotations are optional and describe the service registration or behavior. For example, the tutorial uses the `@Named` tag, because the `test.jsp` and `test.php` need a named reference.

Scripting beans use the `@Named` annotation on a CanDI bean for integration with the JSP EL expression language and with PHP. Nonscripting beans do not declare a `@Named` annotation because CanDI uses the service type and binding annotations for matching.

```
package example;

import javax.enterprise.context.ApplicationScoped
import javax.enterprise.inject.Named;

@ApplicationScoped
@Named("myService")
public class MyServiceBean implements MyService {
  private String _message = "default";

  public void setMessage(String message)
  {
    _message = message;
  }

  public String getMessage()
  {
    return _message;
  }
}
```

Example: MyServiceBean.java

2.1 Using Services from PHP and JSP

CanDI is designed to integrate closely with scripting languages like PHP and JSP. The scripting languages locate a CanDI service or resource using a string,

because scripting lacks the strong typing needed for full dependency injection. As mentioned above, the name of a CanDI service is declared by the `@Named` anntation on the bean itself. The PHP or JSP code will use the name to obtain a reference to the bean. For PHP, the function call is `java_bean` as follows:

```
<?php

$myService = java_bean("myService");

echo $myService->getMessage();

?>
```

Example: test.php

While PHP has a function access to the CanDI service or resource, JSP and JSF grab the CanDI bean with using the JSP expression language. Any CanDI bean with a `@Named` annotation automatically becomes available to EL expressions as follows:

```
message: ${myService.message}
```

Example: test.jsp

3 Resource XML Configuration Pattern

Resources like databases, and queues fit multiple roles in an application and need configuration and description beyond their generic `DataSource` and `BlockingQueue` APIs. While services are generally unique and can use the `@Current` binding, resources will generally create custom `@BindingType` annotations to identify and document the resource.

CanDI encourages a small number of binding annotations used as adjectives to describe resources. A typical medium application like a mail list manager might use half a dozen custom binding adjectives, and may need fewer or more depending on the number of unique resources. Each database, queue, mail, and JPA EntityManager will generally have a unique name. If users need customization and configuration of internal resources, you may need additional binding types. If the application has a single database, it might only have one binding annotation, or might even use `@Current` .

The purpose of the binding annotation is to self-document the resource in the client code. If the application uses `@ShoppingCart` database and a `@ProductCatalog` database, the client code will bind by their description. The code declares its dependencies in a readable way, and lets CanDI and the configuration provide the resource it needs.

The tutorial has `@Red` resource, configured in XML because the user might need to customize the configuration. The resource client, `SetServlet` , uses the adjective annotation in the field declaration as follows:

```
public class SetServlet extends HttpServlet {
  private @Red MyResource _red;
  private @Blue MyResource _blue;

  ...
}
```

Example: SetServlet.java

The XML is short and meaningful, because it's only required for customization, not for wiring and binding. Databases and JMS queues will need to configure the database driver and add the binding adjective. Applications resources can also be configured in XML if exposing the configuration is useful to your users, but unique internal classes like most services will stay out of the XML. In our example, we let the users configure the `data` field of the resource and let them choose the implementation class.

The XML configuration for a bean needs three pieces of data: the driver class, the descriptive binding annotation, and any customization data. Because the driver is the most important, CanDI uses the class as the XML tag and uses the package as the XML namespace. While scanning the XML, the driver class is top and prominent, reflecting its importance. In the example, <`example:BlueResourceBean`> is the driver class.

```
<example:BlueResourceBean xmlns:example="urn:java:example">
  ...
</example:BlueResourceBean>
```

Example: BlueResourceBean instance configuration

In CanDI, the binding annotation is also an XML tag, represented by its classname and package. In CanDI, classes and annotations get XML tags with camel-case names matching the classname, and XML for properties are lower case. The case distinguishes annotations from properties in the XML, improving XML readability.

```
<example:Blue xmlns:example="urn:java:example"/>
```

Example: @Blue annotation configuration

Properties of a resource use the standard beans-style names, so <`example:data`> sets the bean's `setData` property. CanDI converts the XML string value to the property's actual value. In this case, the conversion is trivial,

but CanDI can convert to integers, doubles, enumerations, classes, URLs, etc. Beans have all the configuration capabilities as Resin beans in the resin.xml and resin-web.xml, because Resin uses CanDI for its own internal configuration.

```
<web-app xmlns="http://caucho.com/ns/resin"
         xmlns:example="urn:java:example">

  <example:BlueResourceBean>
    <example:Blue/>
    <example:data>blue resource</example:data>
  </example:BlueResourceBean>

  <example:RedResourceBean>
    <example:Red/>
    <example:data>red resource</example:data>
  </example:RedResourceBean>

</web-app>
```

Example: resin-web.xml

Binding types should generally be descriptive adjectives, so it can describe the injection clearly. Anyone reading code should understand immediately which resource it's using. The tutorial's `@Blue` binding annotation itself is a normal Java annotation marked by a CanDI `@BindingType` annotation. Because of its importance and because there are only a small number of custom annotations, it's important to spend time choosing a good descriptive name for the annotation.

```
package example;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;
import java.lang.annotation.*;
import javax.enterprise.inject.BindingType;

@BindingType
@Documented
@Target({TYPE, METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Blue {
}
```

Example: Blue.java

The resource implementation itself is straightforward. When the resource is a singleton, it will need a `@ApplicationScoped` annotation, just like a service. By default, CanDI will inject a new instance of the bean at every injection point.

```
package example;

public class BlueResourceBean {
  private String _data;

  public void setData(String data)
  {
    _data = data;
  }
}
```

Example: BlueResourceBean.java

4 Startup Pattern

The `@Startup` annotation marks a bean as initializing on server startup. Because the startup bean is discovered through classpath scanning like the other beans, the initialization is controlled by the startup class itself. In other words, looking at the startup class is sufficient, because it doesn't rely on XML for startup. The startup bean uses the `@PostConstruct` annotation on an initialization method to start initialization code.

```
package example;

import javax.annotation.PostConstruct;
import javax.ejb.Startup;
import javax.enterprise.inject.Current;

@Startup
public class MyStartupBean {
  private @Current MyService _service;

  @PostConstruct
  public void init()
  {
    _service.setMessage(this + ": initial value");
  }
}
```

Example: MyStartupBean.java

5 Plugin/Extension Pattern

A plugin or extension architecture can make an application more flexible and configurable. For example, a filtering system, or blueprints or custom actions can add significant power to an application. The plugin pattern uses CanDI's discovery system to find all implementations of the plugin interface.

The `Instance` iterator together with the special `@Any` binding annotation gives all implementations of a resource.

The CanDI `Instance` interface has two uses: return a unique instance programmatically with the `get()` method, and list all instances for a plugin capability. Since `Instance` implements the JDK's `Iterable` interface, you can use it in a `for` loop. Each returned instance obeys the standard CanDI scoping rules, either returning the single value for `@ApplicationScoped` singletons, or creating a new instance for the default.

The `@Any` annotation works with `Instance` to select all values. Because bindings default to the `@Current` binding type, we need to override the default to get all instances.

```
package example;

import javax.enterprise.inject.Any;
import javax.enterprise.inject.Instance;
...

public class GetServlet extends HttpServlet {
  @Any Instance<MyResource> _resources;

  public void service(HttpServletRequest request,
                      HttpServletResponse response)
    throws IOException, ServletException
  {
    PrintWriter out = response.getWriter();

    for (MyResource resource : _resources) {
      out.println("resource: " + resource);
    }
  }
}
```

Example: GetServlet.java