Web Beans: Java Contexts
and Dependency Injection

The new standard for dependency injection and contextual state management

Gavin King
JSR-299 specification lead
Red Hat Middleware LLC

Pete Muir
Web Beans (JSR-299 Reference Implementation) lead
Red Hat Middleware LLC

David Allen
Traducción en italiano: Nicola Benaglia, Francesco Milesi
Spanish Translation: Gladys Guerrero
Red Hat Middleware LLC
Korean Translation: Eun-Ju Ki,

Red Hat Middleware LLC
Traditional Chinese Translation: Terry Chuang
Red Hat Middleware LLC
Simplified Chinese Translation: Sean Wu
Kava Community

Note

JSR-299 has recently changed its name from "Web Beans" to "Java Contexts and Dependency Injection". The reference guide still refers to JSR-299 as "Web Beans" and the JSR-299 Reference Implementation as the "Web Beans RI". Other documentation, blogs, forum posts etc. may use the new nomenclature, including the new name for the JSR-299 Reference Implementation - "Web Beans".

You'll also find that some of the more recent functionality to be specified is missing (such as producer fields, realization, asynchronous events, XML mapping of EE resources).

Parte I. Uso de objetos contextuales

La especificación de Web Beans (JSR-299) define una serie de servicios para el entorno de Java EE que facilitan el desarrollo de aplicaciones. Web Beans entrecruza un ciclo de vida mejorado y un modelo de interacción en tipos de componentes existentes de Java incluyendo los componentes JavaBeans y Enterprise Java Beans. Como complemento para el modelo tradicional de programación Java EE, el servicio de Web Beans proporciona servicios:

- un ciclo de vida mejorada para componentes con estado, vinculados a los *contextos* bien definidos,
- un método typesafe para *inyección de dependencia*,
- interacción a través de un servicio de *notificación de eventos*, y
- un mejor método para vincular *interceptores* a componentes, junto con una nueva clase de interceptor, llamado un *decorador*, el cual es más apropiado para resolver problemas de negocios.

Inyección de dependencia, junto con la administración de ciclo de vida contextual, ahorra al usuario de un API desconocido el tener que hacer y contestar las siguientes preguntas:

- ¿Cuál es el ciclo de vida de este objeto?
- ¿Cuántos clientes simultáneos puede tener?
- ¿Es multihilos?
- ¿En dónde puedo obtener uno?
- ¿Necesito destruirlo explícitamente?
- ¿Dónde debo guardar mi referencia a éste cuando no lo estoy usando directamente?
- ¿Cómo puedo agregar una capa de direccionamiento indirecto, para que la implementación de este objeto pueda variar en el momento de despliegue?
- ¿Cómo hago para compartir este objeto con otros objetos?

Un Web Bean especifica únicamente el tipo y la semántica de otros Web Beans de los que depende. No se necesita conocer el ciclo de vida real, la implementación concreta, el modelo de hilos u otros clientes de cualquier Web Bean a la que dependa. Mejor aún, la implementación concreta, el ciclo de vida y el modelo de hilos de un Web Bean al que éste depende pueden variar según el escenario de despliegue, sin afectar a ningún cliente.

Los eventos, los interceptores y los decoradores mejoran el *acoplamiento-flexible*inherente en este modelo:

- La *notificación de eventos* separa a los productores de eventos de los consumidores,
- los *interceptores* separan las cuestiones técnicas de la lógica de negocios, y
- los *decoradores* permiten la compartimentación de las cuestiones de negocios.

Más importante, Web Beans ofrece todos los servicios en forma*typesafe*. Web Beans nunca utiliza identificadores de cadena para determinar cómo los se ajustan entre sí objetos de colaboración. Además, aunque XML sigue siendo una opción, muy rara vez se utiliza. En su lugar, Web Beans usa la información de teclado disponible en el modelo de objeto Java junto con un nuevo patrón, llamado *anotaciones de enlace*, para conectar a Web Beans, las dependencias, los interceptores y decoradores y sus consumidores de evento.

Los servicios Web Beans son generales y se aplican a los siguientes tipos de componentes existentes en el entorno de Java EE.

- todos los JavaBeans,
- todos los EJB, y
- todos los Servlets.

Web Beans incluso provee los puntos de integración necesarios para que otras clases de componentes definidos por especificaciones de Java EE futuras, hagan uso de los servicios de Web Beans e interactúen con otra clase de Web Bean.

Una gran cantidad de marcos existentes de Java, incluyendo Seam, Guice y Spring, influyeron en Web Beans. No obstante, Web Beans tiene su propio carácter distintivo: más typesafe que Seam, más con estado y menos centrada en XML que Spring, red y aplicación empresarial más capaz que Guice.

Lo más importante es que Web Beans es un JCP estándar que se integra sin problemas con Java EE y con cualquier entorno SE dónde EJB Lite incrustado esté disponible.

Comenzando a escribir Web Beans

Entonces, ¿está preparado para empezar a escribir su primer Web Bean? O quizás está escéptico, preguntándose por qué tipos de arcos le hará ¡saltar la especificación de Web Beans! La buena noticia es que probablemente ya ha escrito y utilizado cientos, quizás miles de Web Beans. Podría no recordar incluso el primer Web Bean que escribió.

1.1. Su primer Web Bean

Con determinadas excepciones, muy especiales, toda clase de Java con un constructor que no acepte parámetros es un Web Bean. Esto incluye cada JavaBean. Además, cada sesión estilo EJB 3 es un Web Bean. Por supuesto, los JavaBeans y EJB que usted ha escrito a diario no han podido aprovechar los nuevos servicios definidos por la especificación de Web Beans, pero podrá utilizar cada uno de ellos como Web Beans # inyectándolos en otros Web Beans, configurándolos a través de los servicios de configuración, incluso agregándoles interceptores y decoradores # sin tocar su código existente.

Suponga que tenemos dos clases existentes de Java, las cuales hemos estado utilizando por años en varias aplicaciones. La primera clase analiza una cadena en un lista de oraciones:

```
public class SentenceParser {
   public List<String
> parse(String text) { ... }
}
```

La segunda clase existente es un bean de front-end sin estado de sesión capaz de traducir oraciones de un idioma a otro:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Donde `Translator` es la interfaz local:

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Lamentablemente, no tenemos una clase preexistente que traduzca todos los documentos de texto. Entonces, escribamos un Web Bean que realice esta tarea:

```
public class TextTranslator {

   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Initializer
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
   }

   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
         sb.append(sentenceTranslator.translate(sentence));
      }
      return sb.toString();
   }

}
```

Podemos obtener una instancia de `TextTranslator` inyectándola en una Web Bean, Servlet o EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
   this.textTranslator = textTranslator;
}
```

De modo alterno, podemos obtener una instancia llamando directamente un método del administrador de Web Bean:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

Pero espere: ¡`TextTranslator` no tiene un constructor sin parámetros! ¿Es éste aún un Web Bean? Bueno, una clase que no tiene un constructor sin parámetros aún puede ser un Web Bean si tiene un constructor anotado `@Initializer`.

Como pudo adivinar, la anotación `@Initializer` tiene algo que ver con la ¡inyección de dependencia! `@Initializer` puede aplicarse a un constructor o método de un Web Bean, y pide a un administrador de Bean llamar a ese constructor o método cuando inicia el Web Bean. El administrador de Web Bean inyectará otros Web Beans a los parámetros del constructor o método.

En el momento de inicialización, el administrador de Web Bean debe confirmar que exista exactamente un Web Bean que complete cada punto de inyección. En nuestro ejemplo, si no estaba disponible ninguna implementación de `Translator` # si el EJB de `SentenceTranslator` no estaba desplegado # el administrador de Web Bean produciría una `UnsatisfiedDependencyException`. Si más de una implementación de `Translator` estuviera disponible, el administrador de Web Bean produciría una `AmbiguousDependencyException`.

1.2. ¿Qué es un Web Bean?

Entonces, ¿qué es, *exactamente* un Web Bean?

Un Web Bean es una clase de aplicación que contiene lógica de negocios. Un Web Bean puede llamarse directamente desde el código de Java, o invocarse a través de Unified EL. Un Web Bean puede acceder recursos transaccionales. Las dependencias entre Web Beans son manejadas automáticamente por el administrador de Web Bean. La mayoría de Web Beans son *con estado* y *contextuales*. El ciclo de vida de un Web Bean siempre es manejado por el administrador de Web Bean.

Volvamos atrás por un segundo. ¿Qué significa "contextual"? Puesto que Web Beans puede tener estados, es importante saber *qué* instancia de bean se tiene. A diferencia de un modelo de componente sin estado (por ejemplo, beans sin estado de sesión) o un modelo de componente singleton (como servlets, o beans singleton), clientes diferentes de un Web Bean ven el Web Bean en estados diferentes. El estado cliente-visible depende de la instancia de Web Bean a la que se refiere el cliente.

No obstante, como un modelo sin estado o un modelo singleton, pero *a diferencia* de los beans con estado de sesión, el cliente no controla el ciclo de vida de la instancia explícitamente creando y destruyéndolo. En su lugar, el *ámbito* del Web Bean determina:

- el ciclo de vida de cada instancia del Web Bean y
- los clientes que comparten una referencia a una instancia determinada del Web Bean.

Para un subproceso dado en una aplicación de Web Beans, puede haber un *contexto activo* asociado con el ámbito del Web Bean. Este contexto puede ser único para el subproceso (por ejemplo, si el Web Bean tiene un ámbito de petición), o puede compartirse con algunos subprocesos (por ejemplo, si el Web Bean tiene un ámbito de sesión) o incluso con todos los otros subprocesos (si es el ámbito de la aplicación).

Los clientes (por ejemplo, otros Web Beans) ejecutando en el mismo contexto verán la misma instancia del Web Bean. Pero los clientes en un contexto diferente verán una instancia diferente.

Una gran ventaja del modelo contextual es que permite a los Web Beans con estado ser tratados como ¡servicios! El cliente no necesita preocuparse por manejar el ciclo de vida del Web Bean que está utilizando, *ni necesita saber qué ciclo de vida es.* Los Web Beans interactúan pasando mensajes, y las implementaciones del Web Bean definen el ciclo de vida de su propio estado. Los Web Beans están en parejas sueltas porque:

- interactúan a través de API públicas bien-definidas
- sus ciclos de vida son completamente dispares

Podemos remplazar un Web Bean por un Web Bean diferente que implemente la misma API y tenga un ciclo de vida diferente (un ámbito diferente) sin afectar la otra implementación de Web Bean. De hecho, Web Beans define una facilidad altamente desarrollada para anular las implementaciones de Web Bean en el momento del despliegue, como también ver en *Sección 4.2, "Tipos de despliegue"*.

Observe que todos los clientes de una Web Bean son Web Beans. Otros objetos tales como Servlets o Message-Driven Beans # los cuales son por naturaleza no inyectables, objetos contextuales # también pueden obtener referencias a Web Beans por inyección.

Más formalmente, de acuerdo con la especificación:

> Un Web Bean comprende:
>
> - Conjunto (no vacío) de Tipos API
> - Un conjunto (no vacío) de tipos de anotación
> - Un ámbito
> - Un tipo de despliegue
> - Alternativamente, un nombre de Web Bean
> - Un conjunto de tipos de interceptor de enlace
> - Una implementación de Web Bean

Veamos lo que significan algunos de estos términos, para el desarrollador de Web Bean.

1.2.1. Tipos API, tipos de enlace e inyección de dependencia

Los Web Beans suelen adquirir referencias a otros Web Beans a través de la inyección de dependencia. Cualquier atributo inyectado especifica un "contrato" que debe cumplir el Web Bean que va a ser inyectado. El contrato es:

- Un tipo API, junto con
- un conjunto de tipos de enlace.

Una API es una clase o interfaz de usuario-definida. (Si el Web Bean es un bean EJB de sesión, el tipo API es la vista de bean de interfaz `@Local` o de clase). Un tipo de enlace representa alguna semántica visible de cliente cumplida por algunas implementaciones de API y no por otras.

Los tipos de enlace están representados por anotaciones de usuario-definidas hechas por ellas mismas `@BindingType`. Por ejemplo, el siguiente punto de inyección tiene un tipo de `PaymentProcessor` de API y un tipo de enlace `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

Si no está explícito ningún tipo de enlace en el punto de inyección, se asumirá el tipo de enlace predeterminado `@Current`.

Para cada punto de inyección, el administrador de Web Bean busca un Web Bean que cumpla el contrato (implemente el API, y tenga todos los tipos de enlace), e inyecta ese Web Bean.

El siguiente Web Bean tiene el tipo de enlace `@CreditCard` e implementa el tipo API `PaymentProcessor`. Podría por lo tanto ser inyectado en el punto de inyección de ejemplo:

```
@CreditCard
public class CreditCardPaymentProcessor
   implements PaymentProcessor { ... }
```

Si un Web Bean no especifica explícitamente un conjunto de tipos de enlace, tiene exactamente un tipo de enlace: el tipo de enlace predeterminado `@Current`.

Web Beans define un *algoritmo de resolución* altamente desarrollado e intuitivo que ayuda al contenedor a decidir qué debe hacer si hay uno más de un Web Beans que cumpla un contrato determinado. Veremos esta información en detalle en *Capítulo 4, Inyección de dependencia*.

1.2.2. Tipos de despliegue

Los *tipos de despliegue* nos permiten clasificar nuestros Web Beans por escenario de despliegue. Un tipo de despliegue es una anotación que representa un escenario de despliegue determinado, por ejemplo, `@Mock`, `@Staging` o `@AustralianTaxLaw`. Aplicamos la anotación a Web Beans la cual debe ser desplegada en ese escenario. Un tipo de despliegue permite a todo un conjunto de Web Beans ser condicionalmente desplegado, con sólo una línea de configuración.

Muchos Web Beans sólo utilizan el tipo de despliegue predeterminado `@Production`, en cuyo caso no se necesita especificar ningún tipo de despliegue. Todos los tres Web Bean en nuestro ejemplo tienen un tipo de despliegue `@Production`.

En un entorno de prueba, podríamos desear remplazar el `SentenceTranslator` Web Bean por un "mock object":

```
@Mock
public class MockSentenceTranslator implements Translator {
   public String translate(String sentence) {
      return "Lorem ipsum dolor sit amet";
   }
}
```

Habilitaremos un tipo de despliegue `@Mock` en nuestro entorno de prueba, para indicar que `MockSentenceTranslator` y cualquier otro Web Bean anotado `@Mock` debería utilizarse.

Hablaremos más acerca de esta característica única y poderosa en *Sección 4.2, “Tipos de despliegue”*.

1.2.3. Ámbito

El *ámbito* define el ciclo de vida y visibilidad de instancias del Web Bean. El modelo de contexto de Web Beans es extensible, acomodando los ámbitos arbitrarios. Sin embargo, ciertos ámbitos importantes son incorporados en la especificación y provistos por el administrador de Web Bean. Un ámbito está representado por un tipo de anotación.

Por ejemplo, cualquier aplicación de red puede tener una *sesión en ámbito* de Web Beans:

```
@SessionScoped
public class ShoppingCart { ... }
```

Una instancia de una sesión en ámbito Web Bean está vinculada a una sesión de usuario y es compartida por todos los solicitantes que ejecutan en el contexto de esa sesión.

Por defecto, Web Beans pertenece a un ámbito especial llamado el *ámbito seudo dependiente*. Web Beans con este ámbito son objetos puros dependientes del objeto en el que son inyectados y su ciclo de vida está vinculado al ciclo de vida del objeto.

Hablaremos más acerca de ámbitos en *Capítulo 5, Ámbitos y contextos*.

1.2.4. Nombres de Web Beans y EL unificado

Un Web Bean puede tener un *nombre*, que le permita ser utilizado en expresiones EL unificadas. Es fácil especificar el nombre de un Web Bean:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Ahora podemos utilizar el Web Bean en cualquier página JSF o JSP:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable
>
```

Es aún más fácil dejar el nombre predeterminado por el administrador de Web Bean:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

En este caso, el nombre predetermina al `shoppingCart` # el nombre de clase no calificado, con el primer caracter cambiado a minúsculas.

1.2.5. Tipos de interceptor de enlace

Web Beans admite la funcionalidad de interceptor definida por el EJB 3, no sólo por beans EJB, sino también por clases de Java comunes. Además, Web Beans proporciona un nuevo método para enlazar interceptores de enlace a beans EJB y otras Web Beans.

Es posible especificar directamente la clase de interceptor a través de la anotación `@Interceptors`:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

Sin embargo, no es más elegante y mejor práctica, llevar indirectamente al interceptor enlazando a través de un *tipo de interceptor de enlace*:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

Hablaremos acerca de los interceptores y decoradores de Web Beans en *Capítulo 7, Interceptores* y *Capítulo 8, Decoradores*.

1.3. ¿Qué clase de objetos pueden ser Web Beans?

Ya hemos visto que JavaBeans, EJB y algunas otras clases de Java pueden ser Web Beans. Pero, exactamente, ¿qué clase de objetos son los Web Beans?

1.3.1. Web Beans sencillos

La especificación de Web Beans dice que una clase de Java concreta es un Web Bean *sencillo* si:

- no es un componente de contenedor EE - administrado, como un EJB, un Servlet o una entidad JPA,
- no es una clase interna no estática,
- no es un tipo en parámetros, y
- tiene un constructor sin parámetros o un constructor `@Initializer` anotado.

Entonces, casi cada JavaBean es un Web Bean sencillo

Cada interfaz implementada directamente o indirectamente por un Web Bean sencillo es un tipo API de un Web Bean sencillo. La clase y superclase también son tipos API.

1.3.2. Web Beans de empresa

La especificación dice que todos los beans de sesión estilo EJB 3- y singleton son *empresariales*. Los mensajes de beans no son Web Beans # porque no están diseñados para ser inyectados en otros objetos # pero pueden aprovechar la mayoría de las funcionalidades de los Web Beans, incluyendo la inyección de dependencia y los interceptores.

No toda interfaz local de un Web Bean empresarial tiene un parámetro de tipo comodín o tipo variable, cada una de sus superinterfaces, es un tipo API del Web Bean de empresa. Si el bean EJB tiene una vista local de clase de bean, la clase de bean, y cada una de sus superclases, también es un tipo API.

Los beans con estado de sesión deben declarar un método de eliminación sin parámetros o un método de eliminación anotado `@Destructor`. El administrador de Web Bean llama a este método para destruir la instancia de bean con estado de sesión al final del ciclo de vida. Este método se llama el método*destructor* del Web Bean empresarial.

```
@Stateful @SessionScoped
public class ShoppingCart {

   ...

   @Remove
   public void destroy() {}

}
```

¿Entonces deberíamos utilizar un Web Bean empresarial en lugar del Web Bean sencillo? Bueno, cada vez que necesitemos los servicios de empresa avanzados ofrecidos por EJB, tales como:

- administración de transacciones nivel-método y seguridad,
- gestión de concurrencia,
- pasivación de nivel-instancia para beans con estado de sesión y grupo-instancia para beans sin estado de sesión
- invocación de servicio de red y remoto
- temporizadores y métodos asíncronos,

deberíamos utilizar un Web Bean empresarial. Cuando no necesitemos ninguna de estas cosas, bastará con un Web Bean sencillo.

Muchos Web Beans (incluyendo toda sesión o ámbito de aplicación Web Bean) están disponibles para acceso concurrente. Por lo tanto, la administración de concurrencia proporcionada por EJB3.1 es especialmente útil. La mayor parte de la sesión y el ámbito de la aplicación WebBeans debe ser EJB.

Los Web Beans que guardan referencias a recursos pesados, o mantienen un montón de estado interno se benefician del ciclo de vida de contenedor avanzado - administrado definido por el modelo EJB `@Stateless`/`@Stateful`/`@Singleton`, con el soporte para pasivación y grupo de instancia.

Por último, suele ser evidente cuando la administración de transacción nivel-método, seguridad nivel-método, temporizadores o métodos remotos o asíncronos se requieren.

Suele ser fácil iniciar con un Web Bean sencillo y luego cambiar a un EJB, con sólo añadir una anotación: `@Stateless`, `@Stateful` o `@Singleton`.

1.3.3. Métodos de productor

Un *método de productor* es un método llamado por el administrador de Web Bean para obtener una instancia del Web Bean cuando no exista en el actual contexto. Un método de productor permite a la aplicación tomar el control total del proceso de iniciación, en lugar de dejar la instanciación al administrador de Web Bean. Por ejemplo:

```
@ApplicationScoped
public class Generator {

   private Random random = new Random( System.currentTimeMillis() );

   @Produces @Random int next() {
      return random.nextInt(100);
   }

}
```

El resultado de un método de productor es inyectado como cualquier otro Web Bean.

```
@Random int randomNumber
```

El método de tipo de retorno y todas las interfaces que extiende/implementa directa o indirectamente son tipos API del método del productor. Si el tipo de retorno es una clase, todas las superclases tienen también tipos API.

Algunos métodos de productor retornan objetos que requieren destrucción explícita:

```
@Produces @RequestScoped Connection connect(User user) {
   return createConnection( user.getId(), user.getPassword() );
}
```

Estos métodos de productor pueden definir *métodos desechables*:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

Este método desechable es llamado automáticamente por el administrador de Web Bean al final de la petición.

Hablaremos mucho más acerca de métodos del productor en *Capítulo 6, Métodos de productor*.

1.3.4. endpoints de JMS

Por último, una cola o tópico JMS puede ser un Web Bean. Web Beans libera al desarrollador del tedio de manejar los ciclos de vida de todos los objetos JMS requeridos para enviar mensajes a colas y tópicos. Discutiremos sobre endpoints de JMS en *Sección 13.4, "endpoints JMS"*.

ejemplo de aplicación de red JSF

Ilustremos estas ideas con un ejemplo. Vamos a implementar inicio/cierre de sesión de usuario para una aplicación que utiliza JSF. Primero, definiremos un Web Bean para mantener el nombre de usuario y contraseña escritos durante el inicio de sesión:

```
@Named @RequestScoped
public class Credentials {

   private String username;
   private String password;

   public String getUsername() { return username; }
   public void setUsername(String username) { this.username = username; }

   public String getPassword() { return password; }
   public void setPassword(String password) { this.password = password; }

}
```

Este Web Bean está vinculado al intérprete de comandos de inicio de sesión en el siguiente formulario JSF:

```
<h:form>
   <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
      <h:outputLabel for="username"
>Username:</h:outputLabel>
      <h:inputText id="username" value="#{credentials.username}"/>
      <h:outputLabel for="password"
>Password:</h:outputLabel>
      <h:inputText id="password" value="#{credentials.password}"/>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
   <h:commandButton value="Logout" acion="#{login.logout}" rendered="#{login.loggedIn}"/>
</h:form
>
```

El trabajo real está hecho por una sesión de ámbito Web Bean que mantiene información acerca del usuario actualmente conectado y expone la entidad del `Usuario` a otras Web Beans:

```
@SessionScoped @Named
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   private User user;

   public void login() {

      List<User
> results = userDatabase.createQuery(
         "select u from User u where u.username=:username and u.password=:password")
         .setParameter("username", credentials.getUsername())
         .setParameter("password", credentials.getPassword())
         .getResultList();

      if ( !results.isEmpty() ) {
         user = results.get(0);
      }

   }

   public void logout() {
      user = null;
   }

   public boolean isLoggedIn() {
     return user!=null;
   }

   @Produces @LoggedIn User getCurrentUser() {
      return user;
   }

}
```

Obviamente, `@LoggedIn` es una anotación de enlace:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD})
@BindingType
```

```
public @interface LoggedIn {}
```

Ahora, cualquier otro Web Bean puede fácilmente inyectar al usuario actual:

```
public class DocumentEditor {

    @Current Document document;
    @LoggedIn User currentUser;
    @PersistenceContext EntityManager docDatabase;

    public void save() {
        document.setCreatedBy(currentUser);
        docDatabase.persist(document);
    }

}
```

Esperamos que este ejemplo de una idea del modelo de programación de Web Bean. En el próximo capítulo, estudiaremos más a fondo la inyección de dependencia de Web Beans.

Getting started with Web Beans, the Reference Implementation of JSR-299

The Web Beans is being developed at *the Seam project* [http://seamframework.org/WebBeans]. You can download the latest developer release of Web Beans from the *the downloads page* [http://seamframework.org/Download].

Web Beans comes with a two deployable example applications: `webbeans-numberguess`, a war example, containing only simple beans, and `webbeans-translator` an ear example, containing enterprise beans. There are also two variations on the numberguess example, the tomcat example (suitable for deployment to Tomcat) and the jsf2 example, which you can use if you are running JSF2. To run the examples you'll need the following:

- the latest release of Web Beans,
- JBoss AS 5.0.1.GA, or
- Apache Tomcat 6.0.x, and
- Ant 1.7.0.

3.1. Using JBoss AS 5

You'll need to download JBoss AS 5.0.1.GA from *jboss.org* [http://www.jboss.org/jbossas/downloads/], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/jboss-5.0.1.GA.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Después necesitaremos decirle a Web Beans en dónde está localizado JBoss. Editar `jboss-as/build.properties` y establecer la propiedad `jboss.home`. Por ejemplo:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

To install Web Beans, you'll need Ant 1.7.0 installed, and the `ANT_HOME` environment variable set. For example:

Nota

JBoss 5.1.0 comes with Web Beans built in, so there is no need to update the server.

```
$ unzip apache-ant-1.7.0.zip
$ export ANT_HOME=~/apache-ant-1.7.0
```

Then, you can install the update. The update script will use Maven to download Web Beans automatically.

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Ahora, ¡está listo para desplegar su primer ejemplo!

Sugerencia

The build scripts for the examples offer a number of targets for JBoss AS, these are:

- `ant restart` - despliega el ejemplo en formato explotado
- `ant explode` - actualiza un ejemplo explotado, sin reiniciar el despliegue
- `ant deploy` - despliega el ejemplo en formato JAR comprimido
- `ant undeploy` - quita el ejemplo del servidor
- `ant clean` - borra el ejemplo

Para desplegar el ejemplo numberguess:

```
$ cd examples/numberguess
```

```
ant deploy
```

Start JBoss AS:

```
$ /Application/jboss-5.0.0.GA/bin/run.sh
```

Sugerencia

If you use Windows, use the `run.bat`script.

Espere que despliegue la aplicación, y diviértase en *http://localhost:8080/webbeans-numberguess*!

Web Beans includes a second simple example that will translate your text into Latin. The numberguess example is a war example, and uses only simple beans; the translator example is an ear example, and includes enterprise beans, packaged in an EJB module. To try it out:

```
$ cd examples/translator
ant deploy
```

Espere a que despliegue la aplicación, y ¡visite *http://localhost:8080/webbeans-translator*!

3.2. Using Apache Tomcat 6.0

You'll need to download Tomcat 6.0.18 or later from *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/apache-tomcat-6.0.18.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Next, we need to tell Web Beans where Tomcat is located. Edit `jboss-as/build.properties` and set the `tomcat.home` property. For example:

```
tomcat.home=/Applications/apache-tomcat-6.0.18
```

Sugerencia

The build scripts for the examples offer a number of targets for Tomcat, these are:

- `ant tomcat.restart` - deploy the example in exploded format
- `ant tomcat.explode` - update an exploded example, without restarting the deployment
- `ant tomcat.deploy` - deploy the example in compressed jar format
- `ant tomcat.undeploy` - remove the example from the server
- `ant tomcat.clean` - clean the example

To deploy the numberguess example for tomcat:

```
$ cd examples/tomcat
ant tomcat.deploy
```

Start Tomcat:

```
$ /Applications/apache-tomcat-6.0.18/bin/startup.sh
```

Sugerencia

If you use Windows, use the `startup.bat`script.

Espere que despliegue la aplicación, y diviértase en *http://localhost:8080/webbeans-numberguess*!

3.3. Using GlassFish

TODO

3.4. El ejemplo numberguess

En la aplicación numberguess se le dan 10 intentos para adivinar un número entre 1 y 100. Después de cada intento, se le dirá si es mayor o menor a su número.

El ejemplo de numberguess consta de una cantidad de Web Beans, archivos de configuración y páginas Facelet JSF, empaquetadas como WAR. Empecemos con los archivos de configuración.

Todos los archivos de configuración para este ejemplo están localizados en `WEB-INF/`, el cual está almacenado en `WebContent` en el árbol fuente. Primero, tenemos `faces-config.xml`, en donde le pedimos a JSF que utilice Facelets:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="1.2"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-facesconfig_1_2.xsd">

   <application>
      <view-handler
>com.sun.facelets.FaceletViewHandler</view-handler>
   </application>

</faces-config
>
```

Hay un archivo `web-beans.xml` vacío, el cual marca esta aplicación como una aplicación de Web Beans.

Por último, está `web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">

  <display-name>Web Beans Numbergues example</display-name>

  <!-- JSF -->
```

```
  <servlet>                                                    (1)
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>                                            (2)
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <context-param>                                              (3)
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>

  <session-config>                                             (4)
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

1. Enable and load the JSF servlet
2. Configure requests to `.jsf` pages to be handled by JSF
3. Tell JSF that we will be giving our source files (facelets) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Nota

Whilst this demo is a JSF demo, you can use Web Beans with any Servlet based web framework.

Let's take a look at the Facelet view:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
```

```
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:s="http://jboss.com/products/seam/taglib">


 <ui:composition template="template.xhtml">                                    (1)
   <ui:define name="content">
     <h1>Guess a number...</h1>
     <h:form id="NumberGuessMain">

       <div style="color: red">                                                (2)
         <h:messages id="messages" globalOnly="false"/>
           <h:outputText id="Higher" value="Higher!" rendered="#{game.number gt game.guess
and game.guess ne 0}"/>
         <h:outputText id="Lower" value="Lower!" rendered="#{game.number lt game.guess and
game.guess ne 0}"/>
       </div>

       <div>

         I'm thinking of a number between #{game.smallest} and #{game.bigge(3)st}.
         You have #{game.remainingGuesses} guesses.
       </div>

       <div>
         Your guess:

         <h:inputText id="inputGuess"                                          (4)
                      value="#{game.guess}"
                      required="true"
                      size="3"
                      disabled="#{game.number eq game.guess}">

           <f:validateLongRange maximum="#{game.biggest}"                      (5)
                                minimum="#{game.smallest}"/>
         </h:inputText>

        <h:commandButton id="GuessButton"                                      (6)
                         value="Guess"
                         action="#{game.check}"
                         disabled="#{game.number eq game.guess}"/>
       </div>
       <div>
                    <h:commandButton  id="RestartButton"  value="Reset"  action="#{game.reset}"
immediate="true" />
       </div>
     </h:form>
```

```
    </ui:define>
  </ui:composition>
</html>
```

1. Facelets is a templating language for JSF, here we are wrapping our page in a template which defines the header.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentance changes to make sure they know what range to guess in.
4. This input field is bound to a Web Bean, using the value expression.
5. A range validator is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of range number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the Web Bean.

El ejemplo existe de 4 clases, las primeras dos son tipos de enlace. Primero, hay un tipo de enlace `@Random`, utilizado para inyectar un número aleatorio:

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

También hay un tipo de enlace `@MaxNumber`, utilizado para inyectar el número máximo posible:

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

La clase `Generator` es responsable de crear el número aleatorio, a través de un método de productor. También expone el número máximo posible a través del método de productor:

```
@ApplicationScoped
public class Generator {

   private java.util.Random random = new java.util.Random( System.currentTimeMillis() );
```

```
  private int maxNumber = 100;

  java.util.Random getRandom()
  {
    return random;
  }

  @Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
  }

  @Produces @MaxNumber int getMaxNumber()
  {
    return maxNumber;
  }

}
```

Notará que el `Generador` es una aplicación en ámbito por lo tanto no obtenemos un número aleatorio diferente cada vez.

El Web Bean final en la aplicación es la sesión en ámbito `Juego`.

Notará que hemos utilizado la anotación `@Named`, para poder utilizar el bean a través de EL en la página JSF. Por último, hemos utilizado la inyección de constructor para inicializar el juego con un número aleatorio. Y, claro está, necesitamos decirle al jugador cuando haya ganado, por lo tanto le damos retroalimentación con `FacesMessage`.

```
package org.jboss.webbeans.examples.numberguess;


import javax.annotation.PostConstruct;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.webbeans.AnnotationLiteral;
import javax.webbeans.Current;
import javax.webbeans.Initializer;
import javax.webbeans.Named;
import javax.webbeans.SessionScoped;
import javax.webbeans.manager.Manager;

@Named
@SessionScoped
```

```
public class Game
{
  private int number;

  private int guess;
  private int smallest;
  private int biggest;
  private int remainingGuesses;

  @Current Manager manager;

  public Game()
  {
  }

  @Initializer
  Game(@MaxNumber int maxNumber)
  {
    this.biggest = maxNumber;
  }

  public int getNumber()
  {
    return number;
  }

  public int getGuess()
  {
    return guess;
  }

  public void setGuess(int guess)
  {
    this.guess = guess;
  }

  public int getSmallest()
  {
    return smallest;
  }

  public int getBiggest()
  {
    return biggest;
```

```
  }

  public int getRemainingGuesses()
  {
    return remainingGuesses;
  }

  public String check()
  {
    if (guess
>number)
    {
      biggest = guess - 1;
    }
    if (guess<number)
    {
      smallest = guess + 1;
    }
    if (guess == number)
    {
      FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Correct!"));
    }
    remainingGuesses--;
    return null;
  }

  @PostConstruct
  public void reset()
  {
    this.smallest = 0;
    this.guess = 0;
    this.remainingGuesses = 10;
    this.number = manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random
>(){});
  }

}
```

3.4.1. The numberguess example in Tomcat

The numberguess for Tomcat differs in a couple of ways. Firstly, Web Beans should be deployed as a Web Application library in `WEB-INF/lib`. For your convenience we provide a single jar suitable for running Web Beans in any servlet container `webbeans-servlet.jar`.

Sugerencia

Of course, you must also include JSF and EL, as well common annotations (`jsr250-api.jar`) which a JEE server includes by default.

Secondly, we need to explicitly specify the servlet listener (used to boot Web Beans, and control it's interaction with requests) in `web.xml`:

```
<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

3.4.2. The numberguess example for Apache Wicket

Whilst JSR-299 specifies integration with Java ServerFaces, Web Beans allows you to inject into Wicket components, and also allows you to use a conversation context with Wicket. In this section, we'll walk you through the Wicket version of the numberguess example.

Nota

You may want to review the Wicket documentation at *http://wicket.apache.org/*.

Like the previous example, the Wicket WebBeans examples make use of the `webbeans-servlet` module. The use of the *Jetty servlet container* [http://jetty.mortbay.org/] is common in the Wicket community, and is chosen here as the runtime container in order to facilitate comparison between the standard Wicket examples and these examples, and also to show how the webbeans-servlet integration is not dependent upon Tomcat as the servlet container.

These examples make use of the Eclipse IDE; instructions are also given to deploy the application from the command line.

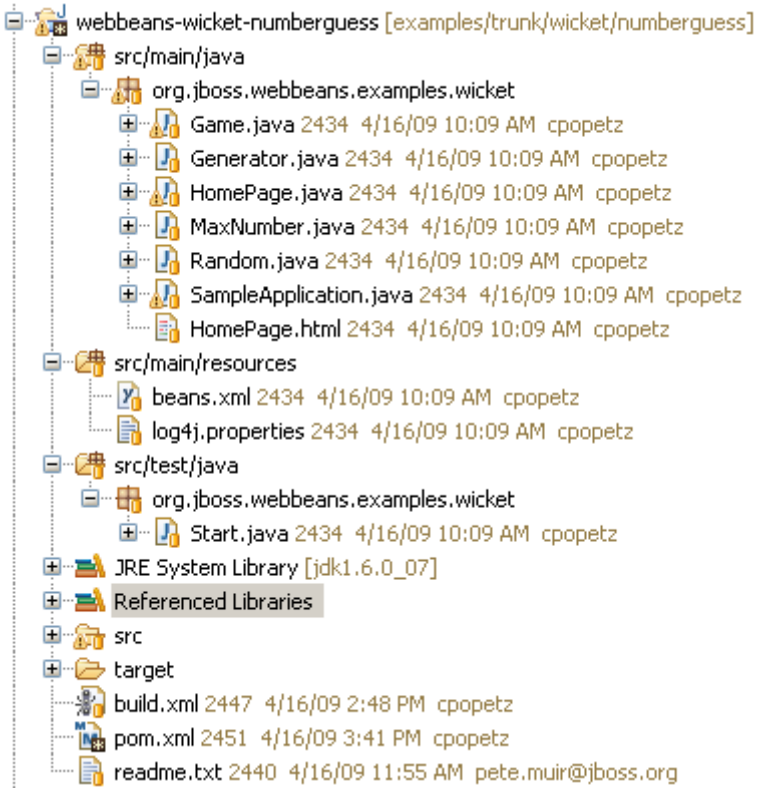
3.4.2.1. Creating the Eclipse project

To generate an Eclipse project from the example:

```
cd examples/wicket/numberguess
mvn -Pjetty eclipse:eclipse
```

Then, from eclipse, choose *File -> Import -> General -> Existing Projects into Workspace*, select the root directory of the numberguess example, and click finish. Note that if you do not intend to

run the example with jetty from within eclipse, omit the "-Pjetty." This will create a project in your workspace called `webbeans-wicket-numberguess`



3.4.2.2. Running the example from Eclipse

This project follows the `wicket-quickstart` approach of creating an instance of Jetty in the `Start` class. So running the example is as simple as right-clicking on that Start class in `src/test/java` in the *Package Explorer* and choosing *Run as Java Application*. You should see console output related to Jetty starting up; then visit able `http://localhost:8080` to view the app. To debug choose *Debug as Java Application*.

3.4.2.3. Running the example from the command line in JBoss AS or Tomcat

This example can also be deployed from the command line in a (similar to the other examples). Assuming you have set up the `build.properties` file in the `examples` directory to specify the location of JBoss AS or Tomcat, as previously described, you can run `ant deploy` from the `examples/wicket/numberguess` directory, and access the application at `http://localhost:8080/webbeans-numberguess-wicket`.

3.4.2.4. Understanding the code

JSF uses Unified EL expressions to bind view layer components in JSP or Facelet views to beans, Wicket defines it's components in Java. The markup is plain html with a one-to-one mapping between html elements and the view components. All view logic, including binding of components to models and controlling the response of view actions, is handled in Java. The integration of

Web Beans with Wicket takes advantage of the same binding annotations used in your business layer to provide injection into your WebPage subclass (or into other custom wicket component subclasses).

The code in the wicket numberguess example is very similar to the JSF-based numberguess example. The business layer is identical!

Differences are:

- Each wicket application must have a `WebApplication` subclass, In our case, our application class is `SampleApplication`:

```
public class SampleApplication extends WebBeansApplication {
  @Override
  public Class getHomePage() {
    return HomePage.class;
  }
}
```

  This class specifies which page wicket should treat as our home page, in our case, `HomePage.class`

- In `HomePage` we see typical wicket code to set up page elements. The bit that is interesting is the injection of the `Game` bean:

```
@Current Game game;
```

  The `Game` bean is can then be used, for example, by the code for submitting a guess:

```
final Component guessButton = new AjaxButton("GuessButton") {
  protected void onSubmit(AjaxRequestTarget target, Form form) {
    if (game.check()) {
```

Nota

All injections may be serialized; actual storage of the bean is managed by JSR-299. Note that Wicket components, like the HomePage and it subcomponents, are *not* JSR-299 beans.

Wicket components allow injection, but they *cannot* use interceptors, decorators and lifecycle callbacks such as `@PostConstruct` or `@Initializer` methods.

- The example uses AJAX for processing of button events, and dynamically hides buttons that are no longer relevant, for example when the user has won the game.
- In order to activate wicket for this webapp, the Wicket filter is added to web.xml, and our application class is specified:

```
<filter>
  <filter-name>wicket.numberguess-example</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>org.jboss.webbeans.examples.wicket.SampleApplication</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>wicket.numberguess-example</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

Note that the servlet listener is also added, as in the Tomcat example, in order to boostrap Web Beans when Jetty starts, and to hook Web Beans into the Jetty servlet request and session lifecycles.

3.4.3. The numberguess example for Java SE with Swing

This example can be found in the `examples/se/numberguess` folder of the Web Beans distribution.

To run this example:

- Open a command line/terminal window in the `examples/se/numberguess` directory
- Ensure that Maven 2 is installed and in your PATH
- Ensure that the `JAVA_HOME` environment variable is pointing to your JDK installation
- execute the following command

```
mvn -Drun
```

There is an empty `beans.xml` file in the root package (`src/main/resources/beans.xml`), which marks this application as a Web Beans application.

The game's main logic is located in `Game.java`. Here is the code for that class, highlighting the changes made from the web application version:

```
@ApplicationScoped                                                    (1) (2)
public class Game implements Serializable
{

   private int number;
   private int guess;
   private int smallest;

   @MaxNumber
   private int maxNumber;

   private int biggest;
   private int remainingGuesses;
   private boolean validNumberRange = true;

   @Current Generator rndGenerator;

   ...

   public boolean isValidNumberRange()
   {
      return validNumberRange;
   }

   public boolean isGameWon()
                                                                      (3)
   {
      return guess == number;
   }

   public boolean isGameLost()
   {
      return guess != number && remainingGuesses <= 0;
   }

   public boolean check()
   {
      boolean result = false;
```

```
        if ( checkNewNumberRangeIsValid() )                    4
        {
            if ( guess > number )
            {
                biggest = guess - 1;
            }

            if ( guess < number )
            {
                smallest = guess + 1;
            }

            if ( guess == number )
            {
                result = true;
            }

            remainingGuesses--;
        }

        return result;
    }

    private boolean checkNewNumberRangeIsValid()
    {
        return validNumberRange = ( ( guess >= smallest ) && ( guess <= biggest ) );
    }

    @PostConstruct
    public void reset()                                        5
    {
        this.smallest = 0;
        ...
        this.number = rndGenerator.next();
    }
}
```

1. The bean is application scoped instead of session scoped, since an instance of the application represents a single 'session'.
2. The bean is not named, since it doesn't need to be accessed via EL

3. There is no JSF `FacesContext` to add messages to. Instead the `Game` class provides additional information about the state of the current game including:

   - If the game has been won or lost
   - If the most recent guess was invalid

   This allows the Swing UI to query the state of the game, which it does indirectly via a class called `MessageGenerator`, in order to determine the appropriate messages to display to the user during the game.
4. Validation of user input is performed during the `check()` method, since there is no dedicated validation phase
5. The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. It cannot use `manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random>(){})` as the JSF example does because there will not be any active contexts like there is during a JSF request.

The `MessageGenerator` class depends on the current instance of `Game`, and queries its state in order to determine the appropriate messages to provide as the prompt for the user's next guess and the response to the previous guess. The code for `MessageGenerator` is as follows:

```
public class MessageGenerator
{
   @Current Game game;                                                    (1)

   public String getChallengeMessage()                                    (2)
   {
      StringBuilder challengeMsg = new StringBuilder( "I'm thinking of a number between " );
      challengeMsg.append( game.getSmallest() );
      challengeMsg.append( " and " );
      challengeMsg.append( game.getBiggest() );
      challengeMsg.append( ". Can you guess what it is?" );

      return challengeMsg.toString();
   }

   public String getResultMessage()                                       (3)
   {
      if ( game.isGameWon() )
      {
         return "You guess it! The number was " + game.getNumber();
```

```
        } else if ( game.isGameLost() )
        {
            return "You are fail! The number was " + game.getNumber();
        } else if ( ! game.isValidNumberRange() )
        {
            return "Invalid number range!";
        } else if ( game.getRemainingGuesses() == Game.MAX_NUM_GUESSES )
        {
            return "What is your first guess?";
        } else
        {
            String direction = null;

            if ( game.getGuess() < game.getNumber() )
            {
                direction = "Higher";
            } else
            {
                direction = "Lower";
            }

            return direction + "! You have " + game.getRemainingGuesses() + " guesses left.";
        }
    }
}
```

1. The instance of `Game` for the application is injected here.
2. The `Game`'s state is interrogated to determine the appropriate challenge message.
3. And again to determine whether to congratulate, console or encourage the user to continue.

Finally we come to the `NumberGuessFrame` class which provides the Swing front end to our guessing game.

```
public class NumberGuessFrame  extends javax.swing.JFrame
{
    private @Current Game game;                                    (1)

    private @Current MessageGenerator msgGenerator;                (2)


    public void start( @Observes @Deployed Manager manager )       (3)
    {
        java.awt.EventQueue.invokeLater( new Runnable()
```

```
        {
            public void run()
            {
                initComponents();
                setVisible( true );
            }
        } );
    }

    private void initComponents() {                                    4

        buttonPanel = new javax.swing.JPanel();
        mainMsgPanel = new javax.swing.JPanel();
        mainLabel = new javax.swing.JLabel();
        messageLabel = new javax.swing.JLabel();
        guessText = new javax.swing.JTextField();
        ...
        mainLabel.setText(msgGenerator.getChallengeMessage());
        mainMsgPanel.add(mainLabel);

        messageLabel.setText(msgGenerator.getResultMessage());
        mainMsgPanel.add(messageLabel);
        ...
    }

    private void guessButtonActionPerformed( java.awt.event.ActionEvent evt )    5
    {
        int guess =  Integer.parseInt(guessText.getText());

        game.setGuess( guess );
        game.check();
        refreshUI();

    }

    private void replayBtnActionPerformed( java.awt.event.ActionEvent evt )    6
    {
       game.reset();
       refreshUI();
    }

    private void refreshUI()                                           7
```

```
    {
        mainLabel.setText( msgGenerator.getChallengeMessage() );
        messageLabel.setText( msgGenerator.getResultMessage() );
        guessText.setText( "" );
        guessesLeftBar.setValue( game.getRemainingGuesses() );
        guessText.requestFocus();
    }

    // swing components
    private javax.swing.JPanel borderPanel;
    ...
    private javax.swing.JButton replayBtn;

}
```

1. The injected instance of the game (logic and state).
2. The injected message generator for UI messages.
3. This application is started in the usual Web Beans SE way, by observing the `@Deployed Manager` event.
4. This method initialises all of the Swing components. Note the use of the `msgGenerator`.
5. `guessButtonActionPerformed` is called when the 'Guess' button is clicked, and it does the following:

    - Gets the guess entered by the user and sets it as the current guess in the `Game`
    - Calls `game.check()` to validate and perform one 'turn' of the game
    - Calls `refreshUI`. If there were validation errors with the input, this will have been captured during `game.check()` and as such will be reflected in the messeges returned by `MessageGenerator` and subsequently presented to the user. If there are no validation errors then the user will be told to guess again (higher or lower) or that the game has ended either in a win (correct guess) or a loss (ran out of guesses).
6. `replayBtnActionPerformed` simply calls `game.reset()` to start a new game and refreshes the messages in the UI.
7. `refreshUI` uses the `MessageGenerator` to update the messages to the user based on the current state of the Game.

3.5. Ejemplo de traductor

El ejemplo de traductor tomará las oraciones que entre y las traducirá en Latín.

El ejemplo de traductor está incorporado como un EAR, y contiene EJB. Como resultado, su estructura es más compleja que el ejemplo de Numberguess.

Nota

EJB 3.1 y Java EE 6 le permiten empaquetar EJB en un WAR, lo cual hará la estructura mucho más simple!

Primero, demos una mirada al agregador EAR, el cual está localizado en el módulo `webbeans-translator-ear`. Maven genera automáticamente la `application.xml`:

```
<plugin>
  <groupId
>org.apache.maven.plugins</groupId>
  <artifactId
>maven-ear-plugin</artifactId>
  <configuration>
    <modules>
      <webModule>
        <groupId
>org.jboss.webbeans.examples.translator</groupId>
        <artifactId
>webbeans-translator-war</artifactId>
        <contextRoot
>/webbeans-translator</contextRoot>
      </webModule>
    </modules>
  </configuration>
</plugin
>
```

Aquí establecemos la ruta de contexto, la cual nos da una url interesante (*http://localhost:8080/webbeans-translator*).

Sugerencia

Si no está utilizando Maven para generar estos archivos, usted necesitaría `META-INF/application.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://
java.sun.com/xml/ns/javaee/application_5.xsd"
```

```
            version="5">
  <display-name
>webbeans-translator-ear</display-name>
  <description
>Ejemplo Ear para la implementaci#n de referencia de JSR 299: Web Beans</
description>

  <module>
    <web>
      <web-uri
>webbeans-translator.war</web-uri>
      <context-root
>/webbeans-translator</context-root>
    </web>
  </module>
  <module>
    <ejb
>webbeans-translator.jar</ejb>
  </module>
</application
>
```

Next, lets look at the war. Just as in the numberguess example, we have a `faces-config.xml` (to enable Facelets) and a `web.xml` (to enable JSF) in `WebContent/WEB-INF`.

Más interesante aún es el facelet utilizado para traducir texto. Al igual que en el ejemplo de Numberguess tenemos una plantilla, la cual rodea el formulario (omitido aquí por razones de brevedad):

```
<h:form id="NumberGuessMain">

  <table>
    <tr align="center" style="font-weight: bold" >
      <td>
        Your text
      </td>
      <td>
        Translation
      </td>
    </tr>
    <tr>
      <td>
      <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80" />
```

```
      </td>
      <td>
        <h:outputText value="#{translator.translatedText}" />
      </td>
    </tr>
  </table>
  <div>
    <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
  </div>

</h:form
>
```

El usuario puede entrar texto en el área de texto a mano izquierda y pulsar el botón de traducir para ver el resultado a la derecha.

Por último, veamos el módulo EJB, `webbeans-translator-ejb`. En `src/main/resources/META-INF` sólo hay un `web-beans.xml` vacío, utilizado para marcar el archivo como si contuviera Web Beans.

Hemos guardado la parte más interesante para el final, ¡el código! El proyecto tiene dos beans sencillos, `SentenceParser` y `TextTranslator` y dos beans empresariales, `TranslatorControllerBean` y `SentenceTranslator`. Por ahora, debe comenzar a familiarizarse con el aspecto de Web Bean, por lo tanto sólo destacaremos aquí las partes más interesantes.

Tanto `SentenceParser` como `TextTranslator` son beans dependientes, y `TextTranslator` utiliza inicialización de constructor:

```
public class TextTranslator {
  private SentenceParser sentenceParser;
  private Translator sentenceTranslator;

  @Initializer
  TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator)
  {
    this.sentenceParser = sentenceParser;
    this.sentenceTranslator = sentenceTranslator;
```

`TextTranslator` es un bean con estado (con una interfaz local de negocios), donde lo mágico sucede - claro está, que no desarrollaramos un traductor completo, ¡pero le dimos una buena luz!

Por último, hay un controlador orientado a UI que recoge el texto desde el usuario y lo envía al traductor. Esta es una petición en ámbito, llamada bean con estado de sesión que inyecta el traductor.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController
{

   @Current TextTranslator translator;
```

El bean también tiene capturadores y configuradores para todos los campos en la página.

Como este es un bean de sesión con estado, tenemos que tener un método de eliminación:

```
   @Remove
   public void remove()
   {

   }
```

El administrador de Web Beans llamará al método de eliminación cuando el bean sea destruido, en este caso al final de la petición.

That concludes our short tour of the Web Beans examples. For more on Web Beans , or to help out, please visit *http://www.seamframework.org/WebBeans/Development*.

Necesitamos ayuda en todas las áreas - corrección de errores, escritura de nuevas funciones, escritura de ejemplos y traducción de esta guía de referencia.

Inyección de dependencia

Web Beans soporta tres mecanismos primarios para inyección de dependencia:

Constructor de inyección de parámetro:

```
public class Checkout {

   private final ShoppingCart cart;

   @Initializer
   public Checkout(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Inyección de parámetro del método *Inicializador*:

```
public class Checkout {

   private ShoppingCart cart;

   @Initializer
   void setShoppingCart(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

E inyección directa de campo:

```
public class Checkout {

   private @Current ShoppingCart cart;

}
```

La inyección de dependencia siempre se presenta cuando la instancia de Web Bean es instanciada primero:

- Primero, el administrador de Web Bean llama al constructor de Web Bean, para obtener una instancia del Web Bean.
- Luego, el administrador de Web Bean inicializa los valores de los campos inyectados del Web Bean.
- Más tarde, el administrador de Web Bean llama a todos los métodos inicializadores del Web Bean.
- Por último, se llama al método de Web Bean `@PostConstruct`, si existe.

La inyección de parámetro constructor no es admitida por beans de EJB, porque EJB es instanciado por el contenedor de EJB, no por el administrador de Web Bean.

Los parámetros de constructores y métodos de inicializador no necesitan ser anotados explícitamente cuando se aplique el tipo de enlace predeterminado `@Current`. Los campos inyectados, sin embargo, *deben* especificar un tipo de enlace, cuando se aplique el tipo de enlace predeterminado. Si el campo no especifica ningún tipo de enlace, no será inyectado.

Los métodos de productor también admiten inyección de parámetro:

```
@Produces Checkout createCheckout(ShoppingCart cart) {
   return new Checkout(cart);
}
```

Por último, los métodos de observador (que encontraremos en *Capítulo 9, Eventos*), los métodos desechables y los métodos destructores, admiten inyección de parámetro.

La especificación de Web Beans define un procedimiento, llamado *algoritmo de resolución de typesafe* que el administrador de Web Bean sigue al identificar el Web Bean para inyectar a un punto de inyección. Este algoritmo parece complejo en un principio, pero una vez que lo entienda, es en realidad muy intuitivo. La resolución de Typesafe se realiza al inicializar el sistema, lo que significa que el administrador informará al usuario inmediatamente si se pueden cumplir las dependencias de un Web Bean, produciendo una `UnsatisfiedDependencyException` o una `AmbiguousDependencyException`.

El propósito de este algoritmo es permitir a múltiples Web Beans implementar el mismo tipo API ya sea:

- permitiendo al cliente seleccionar la aplicación requerida mediante *anotaciones de enlace*,
- permitiendo al desplegador de aplicación seleccionar la aplicación apropiada para una despliegue particular, sin cambios en el cliente, habilitando o inhabilitando los *tipos de despliegue*, o
- permitiendo que una implementación de una API remplace otra implementación de la misma API en el momento del despliegue, sin cambios al cliente, mediante *prioridad de tipo de despliegue*.

Exploremos cómo el administrador de Web Beans determina una Web Bean para ser inyectado.

4.1. Anotaciones de Enlace

Si tenemos más de un Web Bean que implemente un tipo determinado de API, el punto de inyección puede especificar el Web Bean que debe ser inyectado mediante una anotación de enlace. Por ejemplo, deberían haber dos aplicaciones del `PaymentProcessor`:

```
@PayByCheque
public class ChequePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@PayByCreditCard
public class CreditCardPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Donde `@PayByCheque` y `@PayByCreditCard` son anotaciones de enlace:

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

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

Un desarrollador de cliente de Web Bean utiliza la anotación de enlace para especificar exactamente el Web Bean que debe inyectarse.

Uso de inyección de campo:

```
@PayByCheque PaymentProcessor chequePaymentProcessor;
@PayByCreditCard PaymentProcessor creditCardPaymentProcessor;
```

Uso de inyección de método inicializador:

```
@Initializer
public void setPaymentProcessors(@PayByCheque PaymentProcessor
 chequePaymentProcessor,
                          @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

O uso de inyección de constructor:

```
@Initializer
public Checkout(@PayByCheque PaymentProcessor chequePaymentProcessor,
            @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

4.1.1. Anotaciones de enlace con miembros

Las anotaciones de enlace pueden tener miembros:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayBy {
   PaymentType value();
}
```

En cuyo caso, el valor de miembro es importante:

```
@PayBy(CHEQUE) PaymentProcessor chequePaymentProcessor;
@PayBy(CREDIT_CARD) PaymentProcessor creditCardPaymentProcessor;
```

Se puede pedir al administrador de Web Bean que ignore a un miembro de un tipo de anotación de enlace anotando al miembro `@NonBinding`.

4.1.2. Combinaciones de anotaciones de enlace

Un punto de inyección puede incluso especificar múltiples anotaciones de enlace:

```
@Asynchronous @PayByCheque PaymentProcessor paymentProcessor
```

En este caso, sólo el Web Bean que tiene *ambas* anotaciones de enlace sería elegible para inyección.

4.1.3. Anotaciones de enlace y métodos de productor

Incluso los métodos de productor pueden especificar anotaciones de enlace:

```
@Produces
@Asynchronous @PayByCheque
PaymentProcessor createAsyncPaymentProcessor(@PayByCheque PaymentProcessor
 processor) {
   return new AsynchronousPaymentProcessor(processor);
}
```

4.1.4. El tipo de enlace predeterminado

Web Beans define un tipo de enlace `@Current`, el cual es el tipo de enlace predeterminado para cualquier punto de inyección o Web Bean que no especifique explícitamente un tipo de enlace.

Hay dos circunstancias comunes en que se necesita especificar explícitamente a `@Current`:

- en un campo, para declararlo como un campo inyectado con el tipo de enlace por defecto, y
- en un Web Bean, el cual tiene otro tipo de enlace además del tipo de enlace predeterminado.

4.2. Tipos de despliegue

Todos los Web Beans tienen un *tipo de despliegue*. Cada tipo de despliegue identifica un conjunto de Web Beans que debería ser instalado bajo condiciones en algunos despliegues del sistema.

Por ejemplo, podríamos definir un tipo de despliegue llamado `@Mock`, el cual identificaría Web Beans que deben ser instalados sólo cuando el sistema se ejecute dentro de un entorno de pruebas de integración:

```
@Retention(RUNTIME)
 @Target({TYPE, METHOD})
```

```
@DeploymentType
public @interface Mock {}
```

Supongamos que tenemos algunos Web Bean que interactuaron con un sistema externo para procesar pagos:

```
public class ExternalPaymentProcessor {

   public void process(Payment p) {
      ...
   }

}
```

Como este Web Bean no especifica explícitamente un tipo de despliegue, tiene el tipo de despliegue predeterminado `@Production`.

Para prueba de integración o de unidad, el sistema externo está lento o no está disponible. Por lo tanto, creamos el objeto mock:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

   @Override
   public void process(Payment p) {
      p.setSuccessful(true);
   }

}
```

Pero, ¿cómo determina el administrador de Web Bean la aplicación que debe utilizar en un despliegue determinado?

4.2.1. Habilitar tipos de despliegue

Web Beans define dos tipos de despliegue incorporados: `@Production` y `@Standard`. Por defecto, sólo los Web Beans con tipos de despliegue incorporados se habilitan cuando se despliega el sistema. Podemos identificar tipos de despliegue adicionales en un despliegue particular listándolos en `web-beans.xml`.

Volviendo a nuestro ejemplo, cuando desplegamos nuestras pruebas de integración, deseamos que todos nuestros objetos `@Mock` sean instalados:

```
<WebBeans>
   <Deploy>
      <Standard/>
      <Production/>
      <test:Mock/>
   </Deploy>
</WebBeans
>
```

Ahora el administrador de Web Bean identificará e instalará todos los Web Beans anotados `@Production`, `@Standard` o `@Mock` en el momento del despliegue.

El tipo de despliegue `@Standard` es utilizado únicamente por algunos Web Beans especiales definidos por la especificación de Web Beans. No podemos utilizarlo para nuestros propios Web Beans ni inhabilitarlo.

El tipo de despliegue `@Production` es el tipo de despliegue predeterminado para Web Beans que no declaran explícitamente un tipo de despliegue, y que se puede inhabilitar.

4.2.2. Prioridad de tipo de despliegue

Si ha prestado atención, probablemente se estará preguntando cómo escoge Web Bean la aplicación # `ExternalPaymentProcessor` o `MockPaymentProcessor` # . Piense en lo que sucede cuando el administrador encuentra este punto de inyección:

```
@Current PaymentProcessor paymentProcessor
```

Ahora hay dos Web Beans que cumplen el contrato `PaymentProcessor`. Claro está que no podemos utilizar una anotación de enlace para explicar, porque las anotaciones de enlace están codificadas dentro de la fuente en el punto de inyección, y queremos que el administrador pueda decidir en el ¡momento de despliegue!

La solución a este problema es que cada tipo de despliegue tiene una *prioridad* diferente. La prioridad de los tipos de despliegue es determinada por el orden de aparición en `web-beans.xml`. En nuestro ejemplo, `@Mock` es posterior a `@Production` por lo tanto tiene mayor prioridad.

Cada vez que el administrador descubre que más de un Web Bean cumple el contrato (tipo API más anotaciones de enlace) especificado por un punto de inyección, considera la prioridad relativa de los Web Beans. Se escoge el Web Bean que tiene prioridad respecto de los otros para inyectar. Por lo tanto, en nuestro ejemplo, el administrador de Web Bean inyectará `MockPaymentProcessor` al ejecutar en nuestro entorno de prueba de integración (que es precisamente lo que queremos).

Es interesante comparar esta facilidad con las arquitecturas populares del administrador de hoy. Varios contenedores "ligeros" también permiten el despliegue condicional de clases existentes

en el classpath, pero las clases que van a ser desplegadas deben ser explícitamente, listadas de modo individual en el código de configuración o en algún archivo de configuración XML. Web Beans no admite definición de Web Bean ni configuración vía XML, pero en el común de los casos donde no se requiere una configuración compleja, los tipos de despliegue permiten habilitar un conjunto completo de Web Beans con una sóla línea de XML. Mientras tanto, un desarrollador que esté navegando el código puede fácilmente identificar en qué escenarios de despliegue se utilizará el Web Bean.

4.2.3. Ejemplo de tipos de despliegue

Los tipos de despliegue son útiles para toda clase de cosas, algunos ejemplos a continuación:

- Tipos de despliegue `@Mock` y `@Staging` para pruebas
- `@AustralianTaxLaw` para Web Beans de sitio específico
- `@SeamFramework`, `@Guice` para marcos de terceras partes generados en Web Beans
- `@Standard`para Web Beans estándar definidos por la especificación Web Beans

Estamos seguros que puede pensar en más aplicaciones...

4.3. Corregir dependencias insatisfechas

El algoritmo de resolución typesafe falla cuando, después de considerar las anotaciones de enlace y los tipos de despliegue de todos los Web Beans que implementan el tipo API de un punto de inyección, el administrador de Web Bean no puede identificar con precisión un Web Bean para inyectar.

Por lo general es fácil corregir una `UnsatisfiedDependencyException` o una `AmbiguousDependencyException`.

Para corregir una `UnsatisfiedDependencyException`, basta con proporcionar un Web Bean que implemente el tipo API y tenga los tipos de enlace del punto de inyección # o permita el tipo de despliegue de un Web Bean que ya implemente el tipo API y tenga los tipos de enlace.

Para corregir una `AmbiguousDependencyException`, introduzca un tipo de enlace para distinguir entre las dos implementaciones del tipo de API o cambie el tipo de despliegue de una de las implementaciones con el fin de que el administrador de Web Bean pueda utilizar la prioridad de tipo de despliegue para escoger entre ellas. Una `AmbiguousDependencyException` sólo puede presentarse si dos Web Beans comparten un tipo de enlace y tienen exactamente el mismo tipo de despliegue.

Hay algo más que necesita saber cuando utilice inyección de dependencia en Web Beans.

4.4. Los proxy de cliente

Los clientes de un Web Bean inyectado no suelen mantener una referencia directa a una instancia de Web Bean.

Imagine que un Web Bean vinculado al ámbito de aplicación mantiene una referencia directa a un Web Bean vinculado al ámbito de petición. La aplicación en el ámbito de Web Bean es compartida entre muchas peticiones diferentes. No obstante, cada petición ¡debe ver una instancia diferente de la petición en el ámbito de WebBean!

Ahora imaginemos que un enlace de Web Bean a la sesión mantiene una referencia directa a un Web Bean enlazado al ámbito de la aplicación. De vez en cuando, el contexto de sesión se serializa al disco con el fin de utilizar la memoria de un modo más eficiente. Sin embargo, la aplicación en el ámbito de la instancia de Web Bean ¡no debe serializarse junto con la sesión en el ámbito de Web Bean!

Por lo tanto, a menos que un Web Bean tenga un ámbito predeterminado `@Dependent`, el administrador de Web Bean deberá direccionar indirectamente todas las referencias inyectadas al Web Bean a través del objeto de proxy. Este *proxy de cliente* responsable de garantizar que la instancia de Web Bean reciba un método de invocación es la instancia asociada con el contexto actual. El proxy de cliente también permite a los Web Beans vincularse a contextos tales como el contexto de sesión que debe serializarse al disco sin serializar de modo recursivo a otros Web Beans inyectados.

Lamentablemente, debido a limitaciones del lenguaje de Java, el administrador de Web Bean no puede utilizar proxy en algunos tipos de Java. Por lo tanto, el administrador de Web Bean produce un `UnproxyableDependencyException` si no se puede aplicar proxy al tipo de un punto de inyección.

El administrador de Web Bean no puede aplicar proxy en los siguientes tipos de Java:

- las clases que son declaradas `final` o tienen un método `final`,
- las clases que no tienen un constructor no-privado sin parámetros y
- matrices y tipos primarios.

Suele ser muy fácil corregir una `UnproxyableDependencyException`. Basta con añadir un constructor sin parámetros a la clase inyectada, introducir una interfaz, o cambiar el ámbito del Web Bean inyectado a `@Dependent`.

4.5. Obtención de un Web Bean por búsqueda programática

La aplicación puede obtener una instancia de la interfaz `Manager` a través de inyección:

```
@Current Manager manager;
```

El objeto `Manager` proporciona un grupo de métodos para obtener una instancia de Web Bean en forma programática.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class);
```

Las anotaciones de enlace se pueden especificar a través de subclasificaciones de la clase auxiliar `AnnotationLiteral`, porque de otra manera es difícil instanciar un tipo de anotación en Java.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                new AnnotationLiteral<CreditCard
>(){});
```

Si el tipo de vinculación tiene un miembro de anotación, no podemos utilizar una subclase anónima de `AnnotationLiteral` # en su lugar, necesitaremos crear una subclase llamada:

```
abstract class CreditCardBinding
   extends AnnotationLiteral<CreditCard
>
   implements CreditCard {}
```

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                new CreditCardBinding() {
                                   public void value() { return paymentType; }
                                } );
```

4.6. El ciclo de vida de los callbacks, `@Resource`, `@EJB` y `@PersistenceContext`

Los Web Beans de empresa admiten todos los ciclos de vida de las devoluciones de llamadas definidas por la especificación EJB: `@PostConstruct`, `@PreDestroy`, `@PrePassivate` y `@PostActivate`.

Los Web Beans sencillos admiten únicamente las devoluciones de llamadas `@PostConstruct` y `@PreDestroy`.

Web Beans sencillos y empresariales soportan el uso de `@Resource`, `@EJB` y `@PersistenceContext` para inyección de recursos de Java EE, EJB y contextos de persistencia JPA, respectivamente. Web Beans sencillos no admiten el uso de `@PersistenceContext(type=EXTENDED)`.

La devolución de llamada `@PostConstruct` siempre se presenta tras la inyección de todas las dependencias.

4.7. El objeto `InjectionPoint`

Hay algunas clases de objetos # con ámbito `@Dependent` # que necesitan saber algo acerca del objeto o punto de inyección dentro del cual son inyectados para poder hacer lo que hacen. Por ejemplo:

- La categoría de registro para un `Logger` depende de la clase de objeto que lo posea.
- La inyección de un parámetro HTTP o valor de encabezado depende del parámetro o del nombre de encabezado especificado en el punto de inyección.
- La inyección del resultado de una prueba de expresión EL depende de la expresión que fue especificada en el punto de inyección.

Un Web Bean con ámbito `@Dependent` puede inyectar una instancia de `InjectionPoint` y acceder a metadatos relativos al punto de inyección al que pertenezca.

Veamos un ejemplo. El código siguiente es detallado, y vulnerable a problemas de refactorización:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

Este método inteligente de productor permite inyectar un `Logger` JDK sin especificar explícitamente la categoría de registro:

```
class LogFactory {

  @Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
  }

}
```

Ahora podemos escribir:

```
@Current Logger log;
```

¿No está convencido? Entonces, veamos un segundo ejemplo. Para inyectar parámetros, necesitamos definir el tipo de vinculación:

```
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
  @NonBinding public String value();
}
```

Utilizaríamos este tipo de enlace en puntos de inyección, como a continuación:

```
@HttpParam("username") String username;
@HttpParam("password") String password;
```

El siguiente método de productor sí funciona:

```
class HttpParams

  @Produces @HttpParam("")
  String getParamValue(ServletRequest request, InjectionPoint ip) {
    return request.getParameter(ip.getAnnotation(HttpParam.class).value());
  }

}
```

(Observe que el miembro del `valor()` de la anotación `HttpParam` es ignorado por el administrador de Web Bean porque está anotado como `@NonBinding`.)

El administrador de Web Bean proporciona un Web Bean incorporado que implementa la interfaz `InjectionPoint`:

```
public interface InjectionPoint {
  public Object getInstance();
  public Bean<?> getBean();
  public Member getMember():
  public <T extends Annotation
> T getAnnotation(Class<T
> annotation);
  public Set<T extends Annotation
> getAnnotations();
}
```

Ámbitos y contextos

Hasta ahora, hemos visto algunos ejemplos de *anotaciones de tipo ámbito*. El ámbito de un Web Bean determina el ciclo de vida de instancias del Web Bean. El ámbito también determina qué clientes se refieren a qué instancias del Web Bean. Según la especificación de Web Beans, un ámbito determina:

- Cuándo se crea una nueva instancia de un Web Bean con ese ámbito
- Cuándo se destruye una instancia existente de cualquier Web Bean con ese ámbito
- Qué referencias se refieren a una instancia de un Web Bean con ese ámbito

Por ejemplo, si tenemos una sesión con ámbito Web Bean, `UsuarioActual`, todos los Web Beans que son llamados en el contexto de la misma `HttpSession` verán la misma instancia del `UsuarioActual`. Dicha instancia se creará automáticamente la primera vez que se necesite un `UsuarioActual` en esa sesión, y se destruirá automáticamente al terminar la sesión.

5.1. Tipos de ámbito

Web Beans ofrece un *modelo contextual extensible*. Es posible definir nuevos ámbitos creando una nueva anotación de tipo de ámbito:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@ScopeType
public @interface ClusterScoped {}
```

Claro está que esa es la parte fácil. Para que este tipo de ámbito sea útil, necesitaremos también definir un objeto `Contexto` que implemente el ámbito. La implementación de un `Contexto` suele ser una tarea muy técnica, únicamente destinada a desarrollo de marco.

Podemos aplicar un tipo de anotación de ámbito a una clase de implementación de Web Bean para especificar el ámbito del Web Bean:

```
@ClusterScoped
public class SecondLevelCache { ... }
```

Por lo general, se utilizará uno de los ámbitos incorporados de Web Beans.

5.2. Ámbitos incorporados

Web Beans define cuatro ámbitos incorporados:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

Para una aplicación de red que utilice Web Beans:

- cualquier petición de servlet tiene acceso a petición activa, sesión y ámbitos de aplicación, y adicionalmente,
- cualquier petición de JSF tiene acceso a un ámbito de conversación activo.

Los ámbitos de petición y aplicación también están activos:

- durante invocaciones de métodos remotos EJB,
- durante pausas EJB,
- durante la entrega de mensaje a un bean controlado por mensajes, y
- durante invocaciones de servicio de red.

Si la aplicación trata de invocar un Web Bean con un ámbito que no tiene un contexto activo, el administrador de Web Bean produce una `ContextNotActiveException` de Web Bean en tiempo de ejecución.

Tres de los ámbitos incorporados deben ser extremadamente familiares a cualquier desarrollador de Java EE, por eso no perdamos tiempo en explicarlos aquí. No obstante, uno de los ámbitos es nuevo.

5.3. El ámbito de conversación

El ámbito de conversación de Web Beans es un poco parecido al ámbito de sesión tradicional en que mantiene el estado asociado con el usuario del sistema y abarca varias peticiones al servidor. Sin embargo, a diferencia del ámbito de sesión, el ámbito de conversación:

- está demarcado explícitamente por la aplicación, y
- mantiene un estado asociado con una ficha de navegador de red determinada en una aplicación JSF.

Una conversación representa una tarea, una unidad de trabajo desde el punto de vista del usuario. El contexto de conversación mantiene un estado asociado con lo que el usuario está actualmente trabajando. Si el usuario está trabajando en varias tareas al mismo tiempo, habrá múltiples conversaciones.

El contexto de conversación está activo durante cualquier petición de JSF. Sin embargo, la mayoría de las conversaciones se destruyen al final de la petición. Si una conversación debe mantener un estado a través de múltiples peticiones, debe ser explícitamente promovida a *conversación-larga*.

5.3.1. Demarcación de conversación

Web Beans ofrece un Web Bean incorporado para controlar el ciclo de vida de conversaciones en una aplicación JSF. Dicho Web Bean puede obtenerse por inyección:

```
@Current Conversation conversation;
```

Para iniciar la conversación asociada con la petición actual a una conversación larga, llame al método `begin()` desde el código de aplicación. Para programar que el actual contexto de conversación larga se destruya al final de la petición actual, llame a `end()`.

En el ejemplo a continuación, un Web Bean de conversación en ámbito controla la conversación con la que está asociada.

```
@ConversationScoped @Stateful
public class OrderBuilder {

   private Order order;
   private @Current Conversation conversation;
   private @PersistenceContext(type=EXTENDED) EntityManager em;

   @Produces public Order getOrder() {
      return order;
   }

   public Order createOrder() {
      order = new Order();
      conversation.begin();
      return order;
   }

   public void addLineItem(Product product, int quantity) {
      order.add( new LineItem(product, quantity) );
   }

   public void saveOrder(Order order) {
      em.persist(order);
      conversation.end();
```

```
    }

    @Remove
    public void destroy() {}

}
```

Este Web Bean puede controlar su propio ciclo de vida mediante la API de `Conversación`. No obstante, algunos otros Web Beans tienen un ciclo de vida que depende totalmente de otro objeto.

5.3.2. Propagación de conversación

El contexto de conversación se propaga automáticamente con cualquier petición de JSF (presentación de formulario JSF). No se propaga automáticamente con peticiones sin caras, por ejemplo, la navegación mediante un enlace.

Podemos forzar la propagación de una conversación con una petición sin-caras al incluir el identificador único de la conversación como un parámetro de petición. La especificación de Web Beans reserva el parámetro llamado `cid` para este uso. El único identificador de la conversación se puede obtener del objeto `Conversation`, el cual tiene el nombre de Web Beans `conversation`.

Por consiguiente, el siguiente enlace propaga la conversación:

```
<a href="/addProduct.jsp?cid=#{conversation.id}"
>Add Product</a
>
```

El administrador de Web Bean también se requiere para propagar conversaciones a través de cualquier redirección, incluso si la conversación no está marcada como larga. Esto facilita mucho la implementación del patrón de POST-luego-redirigir, sin recurrir a construcciones frágiles tales como un objeto "flash". En este caso, el administrador de Web Bean agrega automáticamente un parámetro a la URL de redirección.

5.3.3. Pausa de conversación

Con el fin de preservar recursos, el administrador de Web Bean puede destruir una conversación y todo el estado en su contexto en cualquier momento. Una implementación del administrador de Web Bean normalmente hace esto con base en alguna clase de pausa # aunque la especificación de Web Beans no lo requiere. La pausa es el periodo de inactividad anterior a la destrucción de la conversación.

El objeto de `Conversación` proporciona un método para configurar el tiempo de espera. Se trata de una ayuda para el administrador de Web Bean quién tiene la libertad de pasar por alto la configuración.

```
conversation.setTimeout(timeoutInMillis);
```

5.4. El seudo ámbito dependiente

Además de los cuatro ámbitos incorporados, Web Beans ofrece el *ámbito seudo dependiente*. Este es el ámbito para el Web Bean que no declare explícitamente un tipo de ámbito.

Por ejemplo, este Web Bean tiene el ámbito de tipo `@Dependent`:

```
public class Calculator { ... }
```

Cuando un punto de inyección de un Web Bean apunta a un Web Bean dependiente, una nueva instancia del Web Bean dependiente es creada cada vez que el primer Web Bean sea instanciado. Las instancias de Web Beans dependientes nunca se comparten entre Web Beans o puntos diferentes de inyección. Ellas son *objetos dependientes* de alguna otra instancia de Web Bean.

Las instancias dependientes de Web Bean se destruyen cuando la instancia de la que dependen es destruida.

Web Beans facilita la obtención de una instancia dependiente de una clase de Java o bean EJB, incluso si la clase o el bean EJB ya se declaró como Web Bean con algún otro tipo de ámbito.

5.4.1. La anotación `@New`

La anotación de enlace incorporada `@New` permite definición *implícita* de un Web Bean dependiente en un punto de inyección. Imagine que declara el siguiente campo inyectado:

```
@New Calculator calculator;
```

Entonces un Web Bean con ámbito `@Dependent`, tipo de enlace `@New`, tipo de API `Calculator`, clase de implementación `Calculator` y tipo de despliegue `@Standard` está definido de modo implícito.

Esto es cierto incluso si `Calculator` está *ya* declarado con un tipo de ámbito diferente, por ejemplo:

```
@ConversationScoped
public class Calculator { ... }
```

Por lo tanto cada uno de los siguientes atributos inyectados obtiene una instancia diferente a `Calculator`:

```
public class PaymentCalc {

    @Current Calculator calculator;
    @New Calculator newCalculator;

}
```

El campo `calculadora` tiene una instancia de conversación-en ámbito de `Calculator` inyectada. El campo `newCalculator` tiene una nueva instancia de `Calculator` inyectada con un ciclo de vida vinculado al propietario de `PaymentCalc`.

La función es particularmente útil con métodos de productor, así como veremos en el siguiente capítulo.

Métodos de productor

Los métodos de productor nos permiten sobrepasar algunas limitaciones que se presentan cuando el administrador de Web Bean, en lugar de la aplicación, es responsable de instanciar objetos. También son la forma más fácil de integrar objetos que no son Web Beans dentro del entorno de Web Beans. (Veremos un segundo método en *Capítulo 12, Definición de Web Beans utilizando XML*.)

Según las especificaciones:

> A Web Beans producer method acts as a source of objects to be injected, where:
>
> - the objects to be injected are not required to be instances of Web Beans,
> - the concrete type of the objects to be injected may vary at runtime or
> - the objects require some custom initialization that is not performed by the Web Bean constructor

For example, producer methods let us:

- expose a JPA entity as a Web Bean,
- expose any JDK class as a Web Bean,
- define multiple Web Beans, with different scopes or initialization, for the same implementation class, or
- vary the implementation of an API type at runtime.

In particular, producer methods let us use runtime polymorphism with Web Beans. As we've seen, deployment types are a powerful solution to the problem of deployment-time polymorphism. But once the system is deployed, the Web Bean implementation is fixed. A producer method has no such limitation:

```
@SessionScoped
public class Preferences {

   private PaymentStrategyType paymentStrategy;

   ...

   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
      switch (paymentStrategy) {
         case CREDIT_CARD: return new CreditCardPaymentStrategy();
```

```
            case CHEQUE: return new ChequePaymentStrategy();
            case PAYPAL: return new PayPalPaymentStrategy();
            default: return null;
        }
    }

}
```

Consider an injection point:

```
@Preferred PaymentStrategy paymentStrat;
```

This injection point has the same type and binding annotations as the producer method, so it resolves to the producer method using the usual Web Beans injection rules. The producer method will be called by the Web Bean manager to obtain an instance to service this injection point.

6.1. Ámbito de un método de productor

El ámbito del método de productor está predeterminado a `@Dependent`, y así será llamado *cada vez* que el administrador de Web Bean inyecte este campo o cualquier otro campo que apunte al mismo método de productor. Así, podría haber múltiples instancias del objeto `PaymentStrategy` para cada sesión de usuario.

Para cambiar esta conducta, podemos agregar una anotación `@SessionScoped` a este método.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
    ...
}
```

Ahora, cuando el método de productor es llamado, el `PaymentStrategy` devuelto se enlazará con el contexto de sesión. El método de productor no será llamado otra vez en la misma sesión.

6.2. Inyección dentro de métodos de productor

No hay un problema en potencia con el código anterior. Las implementaciones de `CreditCardPaymentStrategy` son instanciadas mediante el operador `nuevo` de Java. Los objetos instanciados directamente por la aplicación no pueden hacer uso de la inyección de dependencia y no tienen interceptores.

Si esto no es lo que deseamos podemos utilizar la inyección de dependencia dentro del método del productor para obtener las instancias de Web Bean:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          ChequePaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Espere, ¿qué sucede si `CreditCardPaymentStrategy` es una petición en ámbito de Web Bean? Entonces el método del productor tiene el efecto de "promover" la instancia de petición en ámbito actual dentro del ámbito de sesión. ¡Esto casi seguro es un error! El objeto en ámbito de petición será destruido por el administrador de Web Bean antes de finalizar la sesión, pero la referencia al objeto se dejará "colgando" en el ámbito de sesión. Este error *no* será detectado por el administrador de Web Bean, entonces, ¡por favor tenga un cuidado especial al retornar instancias de Web Bean desde métodos de productor!

Hay por lo menos tres formas de corregir este error. Podemos cambiar el ámbito de la implementación `CreditCardPaymentStrategy`, pero podría afectar a otros clientes de ese Web Bean. Una mejor opción sería cambiar el ámbito del método del productor a `@Dependent` o `@RequestScoped`.

Pero una solución más común es utilizar la anotación especial de enlace `@New`.

6.3. Uso de `@New` con métodos de productor

Considere el siguiente método de productor:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New ChequePaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Entonces una nueva instancia *dependiente* de`CreditCardPaymentStrategy` será creada, pasada al método de productor, devuelta por el método de productor y finalmente vinculada al contexto de sesión. El objeto dependiente no será destruido hasta que el objeto `Preferencias` sea destruido, al final de la sesión.

Parte II. Desarrollo de cÃ³digo de acoplamiento flexible

El primer tema importante de Web Beans es el *acoplamiento flexible*. Ya hemos visto tres medios para lograr dicho acoplamiento:

- Los *tipos de despliegue* habilitan el polimorfismo del tiempo de despliegue,
- los *métodos de productor* habilitan el polimorfismo del tiempo de ejecución, y
- la *administración de ciclo de vida contextual* separa los ciclos de vida de WebBean.

Estas técnicas sirven para habilitar el acoplamiento flexible de cliente y servidor. El cliente ya no está estrechamente ligado a una implementación de un API, ni tiene que administrar el ciclo de vida del objeto del servidor. Este método permite *interactuar a los objetos con estado como si fueran servicios*.

El acoplamiento flexible hace más *dinámico* a un sistema. El sistema puede responder al cambio de una manera bien definida. En el pasado, los marcos que trataban de ofrecer los servicios listados arriba invariablemente lo hacían sacrificando la seguridad. Web Beans es la primera tecnología que logra este nivel de acoplamiento flexible en una forma typesafe.

Web Beans ofrece tres servicios adicionales importantes que amplían el objetivo del acoplamiento flexible:

- los *interceptores* separan las cuestiones técnicas de la lógica de negocios,
- los *decoradores* pueden ser utilizados para separar algunas cuestiones de negocios, y
- las *notificaciones de eventos* separan a los productores de eventos de los consumidores de eventos.

En primer lugar, exploremos los interceptores.

Interceptores

Web Beans reutiliza el interceptor de arquitectura básico de EJB 3.0, extendiendo la funcionalidad en dos direcciones:

- Cualquier Web Bean puede tener interceptores, no sólo beans de sesión.
- Web Beans ofrece un método de anotación más sofisticado para interceptores de enlace a Web Beans.

La especificación EJB define dos clases de puntos de interceptación:

- la interceptación de método de negocios y
- la interceptación de devolución de llamadas de ciclo de vida

Un *interceptor de método de negocios* se aplica a invocaciones de métodos del Web Bean por clientes del Web Bean:

```
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Un *interceptor de devolución de llamadas de ciclo de vida* se aplica a invocaciones de devolución de llamadas de ciclo de vida por el contenedor:

```
public class DependencyInjectionInterceptor {
   @PostConstruct public void injectDependencies(InvocationContext ctx) { ... }
}
```

Una clase de interceptor puede interceptar métodos de devolución de llamadas de ciclo de vida y métodos de negocios.

7.1. Enlaces de interceptor

Suponga que deseamos declarar que algunos de nuestros Web Beans son transaccionales. La primera cosa que necesitamos es una *anotación de enlace de interceptor* para especificar exactamente en cuáles Web Beans estamos interesados:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
```

```
@Retention(RUNTIME)
public @interface Transactional {}
```

Ahora podemos especificar con facilidad que nuestro `ShoppingCart` es un objeto transaccional:

```
@Transactional
public class ShoppingCart { ... }
```

O, si preferimos, podemos especificar que sólo un método es transaccional:

```
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

7.2. Implementación de interceptores

Es maravilloso, pero en alguna parte de la línea vamos a tener que implementar realmente el interceptor que proporciona este aspecto de manejo de transacción. Todo lo que debemos hacer es crear un interceptor estándar EJB, y anotar `@Interceptor` y `@Transactional`.

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Todos los interceptores de Web Beans son Web Beans sencillos, y podemos aprovechar las ventajas de inyección de dependencia y administración de ciclo de vida contextual.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

Múltiples Interceptores pueden utilizar el mismo tipo de vinculación de interceptor.

7.3. Habilitar Interceptores

Por último, necesitamos *habilitar* nuestro interceptor en `web-beans.xml`.

```
<Interceptors>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

¿Por qué el corchete angular permanece?

Bien, la declaración XML resuelve dos problemas:

- nos permite especificar una orden total para todos los interceptores en nuestro sistema, garantizando una conducta determinante y
- nos permite habilitar o inhabilitar clases de interceptores en el momento del despliegue.

Por ejemplo, podemos especificar que nuestro interceptor de seguridad se ejecuta antes que nuestro `TransactionInterceptor`.

```
<Interceptors>
   <sx:SecurityInterceptor/>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

O podemos ¡apagarlo en nuestro entorno de prueba!

7.4. Enlaces de interceptor con miembros

Suponga que deseamos agregar alguna información adicional a nuestra anotación `@Transactional`:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
   boolean requiresNew() default false;
}
```

Web Beans utilizará el valor de `requiresNew` para escoger entre dos interceptores, `TransactionInterceptor` y `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Ahora podemos utilizar `RequiresNewTransactionInterceptor` de esta manera:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

Pero, ¿qué sucede si sólo tenemos un interceptor y queremos que el administrador ignore el valor de `requiresNew` al vincular interceptores? Podemos utilizar la anotación `@NonBinding`:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
   @NonBinding String[] rolesAllowed() default {};
}
```

7.5. Anotaciones de enlace de múltiples interceptores

Generalmente utilizamos combinaciones de tipos de interceptores de enlace para vincular múltiples interceptores a un Web Bean. Por ejemplo, la siguiente declaración sería utilizada para enlazar `TransactionInterceptor` y `SecurityInterceptor` al mismo Web Bean:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

Sin embargo, en casos muy complejos, el mismo interceptor puede especificar una combinación de tipos de interceptor de enlace:

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Entonces este interceptor podría estar vinculado al método `checkout()` mediante cualquiera de las siguientes combinaciones:

```
public class ShoppingCart {
   @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

```
@TransactionI
public class ShoppingCart {
   @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
   public void checkout() { ... }
}
```

7.6. Herencia del tipo de interceptor de enlace

Una limitación de la compatibilidad del lenguaje de Java para anotaciones es la falta de herencia de anotación. En realidad, las anotaciones deberían tener reutilización incorporada, para permitir a este tipo que funcione:

```
public @interface Action extends Transactional, Secure { ... }
```

Bueno, afortunadamente, Web Beans funciona en torno a esta característica de Java. Podemos anotar un interceptor de tipo de enlace con otros tipos de interceptores de enlace. Los enlaces de interceptor son transitivos # cualquier Web Bean con el primer enlace de interceptor hereda los enlaces de interceptor declarados como meta-anotaciones.

```
@Transactional @Secure
```

```
@InterceptorBindingType
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Cualquier Web Bean anotado `@Action` estará vinculado a `TransactionInterceptor` y `SecurityInterceptor`. (E incluso a `TransactionalSecureInterceptor`, si éste existe.)

7.7. Uso de `@Interceptors`

La anotación `@Interceptors` definida por la especificación de EJB es compatible con Web Beans empresariales y sencillos, por ejemplo:

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
   public void checkout() { ... }
}
```

Sin embargo, este enfoque tiene los siguientes inconvenientes:

- la implementación de interceptor está codificada en código de negocios,
- los interceptores no se pueden fácilmente inhabilitar en el momento del despliegue, y
- la orden del interceptor es no-global # es determinada por la orden en que se listan los interceptores en el nivel de clase.

Por lo tanto, recomendamos el uso de Web Beans estilo interceptor de enlaces.

Decoradores

Los interceptores son una forma potente de capturar y distinguir cuestiones que son *ortogonales* al tipo de sistema. Cualquier interceptor puede interceptar invocaciones de cualquier tipo de Java. Esto los hace perfectos para resolver cuestiones técnicas como la administración de transacción y la seguridad. Sin embargo, por naturaleza, los interceptores desconocen la semántica real de los eventos que interceptan. Por lo tanto, los interceptores no son una herramienta apropiada para distinguir cuestiones relacionadas con negocios.

En cambio, los *decoradores* interceptan invocaciones únicamente para una determinada interfaz de Java y por lo tanto, conocen toda la semántica asociada a la interfaz. Esto los hace una herramienta perfecta para representar algunas clases de cuestiones de negocios. También significa que los decoradores no tienen la generalidad de un interceptor. Los decoradores no pueden resolver problemas técnicos que atraviesan muchos tipos dispares.

Supongamos que tenemos una interfaz que representa cuentas:

```
public interface Account {
   public BigDecimal getBalance();
   public User getOwner();
   public void withdraw(BigDecimal amount);
   public void deposit(BigDecimal amount);
}
```

Varios Web Beans diferentes en nuestro sistema implementan la interfaz de `Cuenta`. No obstante, tenemos un requisito legal que, para cualquier clase de cuenta, las transacciones grandes deben ser registradas por el sistema en un registro especial. Este es el trabajo perfecto para un decorador.

Un decorador es un Web Bean sencillo que implementa el tipo que decora y es anotado `@Decorator`.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {

   @Decorates Account account;

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      account.withdraw(amount);
      if ( amount.compareTo(LARGE_AMOUNT)
```

```
>0 ) {
            em.persist( new LoggedWithdrawl(amount) );
        }
    }

    public void deposit(BigDecimal amount);
        account.deposit(amount);
        if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
            em.persist( new LoggedDeposit(amount) );
        }
    }

}
```

A diferencia de otros Web Beans sencillos, un decorador puede ser una clase abstracta. Si no hay nada especial que el decorador tenga que hacer para un método particular de la interfaz decorada, usted no necesita implementar ese método.

8.1. Atributos de delegado

Todos los decoradores tienen un *atributo de delegado*. El tipo y los tipos de vinculación del atributo de delegado determinan los Web Beans a los que el decorador está vinculado. El tipo de atributo de delegado debe implementar o extender todas las interfaces ejecutadas por el decorador.

Este atributo de delegado especifica que el decorador está vinculado a los Web Beans que implementan `Cuenta`:

```
@Decorates Account account;
```

Un atributo de delegado puede especificar una anotación de enlace. Luego el decorador sólo estará vinculado a los Web Beans con el mismo enlace.

```
@Decorates @Foreign Account account;
```

Un decorador está vinculado a cualquier Web Bean que:

- tenga el tipo de atributo de delegado como un tipo API, y
- tenga todos los tipos de vinculación declarados por el atributo de delegado.

El decorador puede invocar el atributo de delegado, el cual tiene casi el mismo efecto que llamar a `InvocationContext.proceed()` desde un interceptor.

8.2. Habilitar decoradores

Necesitamos *habilitar* nuestro decorador en `web-beans.xml`.

```
<Decorators>
   <myapp:LargeTransactionDecorator/>
</Decorators
>
```

Esta declaración sirve para decoradores al igual que la declaración `<Interceptores>` sirve para interceptores:

- nos permite especificar un orden total para los decoradores en nuestro sistema, garantizando una conducta de determinación y
- nos permite habilitar o inhabilitar las clases de decorador en el momento de implementación.

Los interceptores para un método son llamados antes de los decoradores que aplican a ese método.

Eventos

La notificación de eventos de Web Beans permite a Web Beans interactuar de una manera completamente disociada. Los *productores* crean eventos que son enviados luego a *observadores* de evento por el administrador de Web Beans. Este esquema básico podría parecerse al patrón conocido observador/observable, pero hay un par de cambios:

- no solamente los productores de eventos están disociados de los observadores; los observadores están completamente disociados de los productores.
- los observadores pueden especificar una combinación de "selectores" para limitar el conjunto de notificaciones de eventos que recibirán y
- los observadores pueden ser notificados inmediatamente o pueden especificar que la entrega del evento sea retrasada hasta el final de la transacción actual

9.1. Observadores de evento

Un *método de observador* es un método de un Web Bean con un parámetro anotado `@Observes`.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

El parámetro anotado es llamado el *parámetro de evento*. El tipo del parámetro de evento es el *tipo de evento* observado. Los métodos de observador pueden también especificar "selectores", los cuales son sólo instancias de tipos de enlaces de Web Beans. Cuando un tipo de enlace se utiliza como un selector de evento, es llamado un *tipo de enlace de evento*.

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Updated { ... }
```

Especificamos los enlaces de evento del método de observador al anotar el parámetro de evento:

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

Un método de observador no necesita especificar ningún enlace de evento # en este caso está interesado en *todos* los eventos de un tipo determinado. Si no especifica enlaces de eventos, sólo está interesado en eventos que también tienen esos enlaces de eventos.

El método de observador puede tener parámetros adicionales, los cuales se inyectan de acuerdo con la semántica de inyección del parámetro usual de método de Web Beans:

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

9.2. Productores de Evento

El productor de evento puede obtener un objeto que *notifica el evento*por inyección:

```
@Observable Event<Document
> documentEvent
```

La anotación `@Observable` define implícitamente un Web Bean con ámbito `@Dependent` y tipo de despliegue `@Standard`, con una implementación provista por el administrador de Web Bean.

Un productor crea eventos llamando al método `fire()` de la interfaz del `Evento`, pasando un *objeto de evento*:

```
documentEvent.fire(document);
```

Un objeto de evento puede ser una instancia de una clase de Java que no tiene variables de tecla o parámetros de comodines. El evento será entregado a cada método de observador que:

- tenga un parámetro de evento al cual el objeto de evento sea asignable y
- especifique que no hay enlaces de eventos.

El administrador de Web Beans simplemente llama a todos los métodos de observador, pasando el objeto del evento como el valor de un parámetro de evento. Si cualquier método de observador produce una excepción, el administrador de Web Beans se detiene llamando a los métodos de observador y la excepción es reenviada por el método `fire()`.

Para especificar un "selector", el productor del evento puede pasar una instancia del tipo de enlace del evento al método `fire()`:

```
documentEvent.fire( document, new AnnotationLiteral<Updated
>(){} );
```

La `Anotación Literal` clase auxiliar hace posible crear una instancia de tipos de enlaces en línea, ya que de otra manera es difícil hacerlo en Java.

El evento será entregado a cada método de observador que:

- tenga un parámetro de evento al cual el objeto de evento sea asignable y
- no especifique ningún enlace de evento *excepto* para enlaces de evento pasados a `fire()`.

De modo alternativo, se pueden especificar eventos de enlaces anotando el punto de inyección de notificador de eventos:

```
@Observable @Updated Event<Document
> documentUpdatedEvent
```

Luego cada evento disparado vía esta instancia de `Evento` tiene el enlace de evento anotado. El evento será enviado a cada método de observador que:

- tenga un parámetro de evento al cual el objeto de evento sea asignable y
- no especifica ningún enlace de evento*excepto* para los enlaces de eventos pasados a `fire()` o a los enlaces de evento anotados del punto de inyección del notificador de evento.

9.3. Registro dinámico de observadores

Suele ser útil registrar dinámicamente un observador de evento. La aplicación puede implementar la interfaz del `Observador` y registrar una instancia con un notificador de evento llamando el método de `observe()`.

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } } );
```

Los tipos de enlace de evento pueden ser especificados por el punto de inyección del notificador del evento o pasando las instancias de tipo de enlace de evento al método `observe()`:

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } },
                                    new AnnotationLiteral<Updated
>(){} );
```

9.4. Enlaces de evento con miembros

Un tipo de enlace de evento puede tener miembros de anotación:

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
```

```
public @interface Role {
   RoleType value();
}
```

El valor de miembro se utiliza para limitar los mensajes enviados al observador:

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

Los miembros de tipo de enlace de evento se pueden especificar estáticamente por el producto del evento, a través de anotaciones en el punto de inyección del notificador de evento.

```
@Observable @Role(ADMIN) Event<LoggedIn
> LoggedInEvent;}}
```

Alternativamente, el valor del miembro de tipo de enlace de evento puede ser determinado de modo dinámico por el productor de evento. Empezamos por escribir una subclase abstracta de `AnnotationLiteral`:

```
abstract class RoleBinding
   extends AnnotationLiteral<Role
>
   implements Role {}
```

El productor de evento pasa una instancia de esta clase a `fire()`:

```
documentEvent.fire( document, new RoleBinding() { public void value() { return user.getRole();
 } } );
```

9.5. Enlaces de evento múltiples

Los tipos de enlaces de evento pueden combinarse, por ejemplo:

```
@Observable @Blog Event<Document
> blogEvent;
...
if (document.isBlog()) blogEvent.fire(document, new AnnotationLiteral<Updated
>(){});
```

Cuando un evento ocurre, todos los siguientes métodos de observador serán notificados:

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

9.6. Observadores transaccionales

Los observadores transaccionales reciben las notificaciones de sus eventos antes o después de la fase de finalización de la transacción en la se creó el evento. Por ejemplo, el siguiente método de observador necesita actualizar un conjunto de resultados de petición almacenado en caché en el contexto de aplicación, pero sólo cuando las transacciones que actualizan el árbol de `Categoría` tengan éxito:

```
public void refreshCategoryTree(@AfterTransactionSuccess @Observes CategoryUpdateEvent
 event) { ... }
```

Hay tres clases de observadores transaccionales:

- Los observadores`@AfterTransactionSuccess` son llamados tras la fase de finalización de la transacción, pero sólo si la transacción finaliza exitosamente.
- Los observadores `@AfterTransactionFailure` son llamados tras la fase de finalización de la transacción, pero sólo si la transacción no se finaliza con éxito.
- Los observadores `@AfterTransactionCompletion` son llamados tras la fase de finalización de la transacción.
- Los observadores `@BeforeTransactionCompletion` son llamados durante la fase anterior de finalización de la transacción

Los observadores transaccionales son muy importantes en un modelo de objetos con estado como Web Beans, porque el estado suele ser mantenido para más de una transacción atómica.

Imagine que hemos almacenado en caché una serie de resultados de petición JPA en el ámbito de la aplicación:

```
@ApplicationScoped @Singleton
public class Catalog {

   @PersistenceContext EntityManager em;

   List<Product
> products;

   @Produces @Catalog
   List<Product
> getCatalog() {
      if (products==null) {
         products = em.createQuery("select p from Product p where p.deleted = false")
            .getResultList();
      }
      return products;
   }

}
```

De vez en cuando, se crea o borra un `Producto`. Cuando esto ocurre, necesitamos refrescar el catálogo del `Producto`. No obstante, deberíamos esperar hasta *después* de que la transacción finalice exitosamente antes de ¡actualizar!

El Web Bean que crea y borra `Productos` podría crear eventos, por ejemplo:

```
@Stateless
public class ProductManager {

   @PersistenceContext EntityManager em;
   @Observable Event<Product
> productEvent;

   public void delete(Product product) {
      em.delete(product);
      productEvent.fire(product, new AnnotationLiteral<Deleted
>(){});
   }

   public void persist(Product product) {
      em.persist(product);
      productEvent.fire(product, new AnnotationLiteral<Created
>(){});
```

```
    }

    ...

}
```

Ahora el `Catálogo` puede observar los eventos después de finalizar la transacción exitosamente:

```
@ApplicationScoped @Singleton
public class Catalog {

    ...

    void addProduct(@AfterTransactionSuccess @Observes @Created Product product) {
        products.add(product);
    }

    void addProduct(@AfterTransactionSuccess @Observes @Deleted Product product) {
        products.remove(product);
    }

}
```

Parte III. Aprovechar al mÃ¡ximo un tecleado fuerte

El segundo tema importante de Web Beans es el *tecleado fuerte*. La información acerca de dependencias, interceptores y decoradores de Web Bean y la información sobre consumidores de eventos para un productor de eventos, está contenida en construcciones de Java typesafe que pueden ser validadas por el compilador.

No necesita ver los identificadores de cadena en código de Web Beans, no porque el marco los esté escondiendo del uso inteligente de las reglas # llamadas"configuración por convención" # sino porque simplemente ¡no hay cadenas con qué comenzar!

El beneficio evidente de este método es que *cualquier* IDE puede proporcionar autofinalización, validación y refactorización sin necesitar herramientas especiales. Pero hay un segundo beneficio menos evidente. Resulta que cuando empieza a pensar en identificación de objetos, eventos o interceptores a través de anotaciones en lugar de nombres, tendrá la oportunidad de aumentar el nivel semántico de su código.

Web Beans le anima a desarrollar anotaciones que modelen conceptos, por ejemplo,

- `@Asynchronous`,
- `@Mock`,
- `@Secure` o
- `@Updated`,

en lugar de utilizar nombres compuestos como

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,
- `SecurityInterceptor` o
- `DocumentUpdatedEvent`.

Las anotaciones son reutilizables. Ayudan a describir cualidades comunes de partes dispares del sistema. Nos ayudan a categorizar y entender nuestro código. Nos ayudan a tratar las cuestiones comunes en una forma común. Hacer nuestro código más leíble y entendible.

Los *estereotipos* de Web Beans van más allá de este paso. Un estereotipo modela un *rol* común en su arquitectura de aplicación. El estereotipo encapsula varias propiedades del rol, incluyendo ámbito, enlaces de interceptor, tipo de despliegue, etc, en un sólo paquete reutilizable.

Incluso metadatos XML de Web Beans es tecleado ¡fuertemente! No hay compilador para XML, por eso Web Beans aprovecha los esquemas XML para validar los tipos de Java y los atributos que aparecen en XML. Este enfoque hace que el archivo XML sea más leíble, así como las anotaciones lo hicieron con nuestro código de Java.

Ahora estamos listos para conocer otras funciones más avanzadas de Web Beans. Tenga en cuenta que estas funciones hacen a nuestro código más fácil de validar y más entendible. La mayoría del tiempo no se *necesita* realmente utilizarlas, pero si se utilizan de modo inteligente, se llegará a apreciar su poder.

Estereotipos

Según la especificación de Web Beans:

> En muchos sistemas, el uso de patrones arquitecturales produce una serie de roles de Web Beans recurrentes. Un estereotipo permite al desarrollador de marco identificar dicho rol y declarar algunos metadatos comunes para Web Beans con ese rol en un lugar central.
>
> Un estereotipo encapsula cualquier combinación de:
>
> - un tipo de despliegue predeterminado,
> - un tipo de ámbito predeterminado,
> - una restricción en el ámbito de Web Bean,
> - un requisito que implementa el Web Bean o extiende un cierto tipo y
> - una serie de anotaciones de enlace del interceptor.
>
> Un estereotipo puede también especificar que todos los Web Beans con el estereotipo tengan nombres de Web Beans predeterminados.
>
> Un Web Bean puede declarar cero, uno o múltiples estereotipos.

Un estereotipo es un tipo de anotación Java. Dicho estereotipo identifica clases de acción en algún marco MVC:

```
@Retention(RUNTIME)
@Target(TYPE)
@Stereotype
public @interface Action {}
```

Utilizamos el estereotipo aplicando la anotación al Web Bean.

```
@Action
public class LoginAction { ... }
```

10.1. El ámbito predeterminado y el tipo de despliegue para un estereotipo

Un estereotipo puede especificar un ámbito y /o tipo de despliegue predeterminados para Web Beans con ese estereotipo. Por ejemplo, si el tipo de despliegue `@WebTier` identifica Web Beans

que deben ser desplegados sólo cuando el sistema se ejecuta como una aplicación de red, podríamos especificar los siguientes valores por defecto para clases de acción:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype
public @interface Action {}
```

Obviamente, una acción particular aún puede omitir estos valores predeterminados si es necesario:

```
@Dependent @Mock @Action
public class MockLoginAction { ... }
```

Si deseamos forzar todas las acciones a un ámbito determinado, también lo podemos hacer.

10.2. Restricción de ámbito y tipo con un estereotipo

Supongamos que deseamos evitar acciones de declarar determinados ámbitos. Web Beans nos permite especificar explícitamente la serie de ámbitos permitidos para Web Beans con un estereotipo determinado. Por ejemplo:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype(supportedScopes=RequestScoped.class)
public @interface Action {}
```

Si una clase de acción determinada intenta especificar un ámbito diferente al ámbito de la petición de Web Beans, el administrador de Web Bean producirá una excepción en el momento de inicialización.

También podemos forzar todos los Web Bean con un estereotipo determinado para implementar una interfaz o extender una clase:

```
@Retention(RUNTIME)
@Target(TYPE)
```

```
@RequestScoped
@WebTier
@Stereotype(requiredTypes=AbstractAction.class)
public @interface Action {}
```

Si una clase particular de acción no extiende la clase `AbstractAction`, el administrador de Web Bean producirá una excepción en el momento de inicialización.

10.3. Enlaces de interceptor para esterotipos

Un estereotipo puede especificar una serie de enlaces de interceptor para que sean heredados por todos los Web Beans con ese estereotipo.

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@WebTier
@Stereotype
public @interface Action {}
```

¡Esto nos ayuda a obtener cuestiones técnicas aún más allá del código de negocios!

10.4. Predeterminación de nombre con estereotipos

Por último, podemos especificar que todos los Web Beans con un determinado estereotipo tengan un nombre de Web Bean, predeterminado por el administrador del Web Bean. Se suele hacer referencia a las acciones en páginas JSP, así que son un caso de uso perfecto para esta función. Todo lo que necesitamos es agregar una anotación vacía a `@Named`.

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
@WebTier
@Stereotype
public @interface Action {}
```

Ahora, `LoginAction` se llamará`loginAction`.

10.5. Estereotipos estándar

Ya hemos visto dos estereotipos estándar definidos por la especificación de Web Beans: `@Interceptor` y `@Decorator`.

Web Beans define otro estereotipo estándar:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

Este estereotipo está destinado a ser utilizado con JSF. En lugar de usar beans JSF administrados, solamente anote un Web Bean `@Model`, y utilícelo directamente en su página JSF.

Specialization

Hemos visto cómo el modelo de inyección de dependencia de Web Beans nos permite *omitir* la implementación de un API en el momento del despliegue. Por ejemplo, la siguiente Web Bean empresarial provee una implementación del `Procesador de Pago` de API en producción:

```
@CreditCard @Stateless
public class CreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Pero en nuestro entorno, omitimos esa implementación de `PaymentProcessor` con un Web Bean diferente:

```
@CreditCard @Stateless @Staging
public class StagingCreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Lo que hemos tratado de hacer con `StagingCreditCardPaymentProcessor` es remplazar totalmente a `AsyncPaymentProcessor` en un despliegue particular del sistema. En ese despliegue, el tipo de despliegue `@Staging` tendría una prioridad más alta que el tipo de despliegue predeterminado `@Production`, y por ende los clientes con el siguiente punto de inyección:

```
@CreditCard PaymentProcessor ccpp
```

Recibirían una instancia de `StagingCreditCardPaymentProcessor`.

Lamentablemente, hay varias trampas en las que se puede caer fácilmente:

- el Web Bean de prioridad más alta puede que no implemente todos los tipos de API del Web Bean que intenta omitir,
- el Web Bean de prioridad más alta puede que no declare todos los tipos de enlace del Web Bean que intenta omitir,
- el Web Bean de prioridad más alta puede que no tenga el mismo nombre que el Web Bean que intenta omitir, o

- el Web Bean que intenta omitir podría declarar un método de productor, método desechable o método de observador.

En cada uno de estos casos, el Web Bean que hemos tratado de omitir se podría llamar aún en el tiempo de ejecución. Por lo tanto, la omisión de alguna manera tiende a desarrollar error.

Web Beans ofrece una función especial, llamada *Specialization*, la cual ayuda al desarrollador a evitar estas trampas. Specialization parece un poco esotérica al comienzo, pero es fácil de utilizar en la práctica y realmente apreciará la seguridad adicional que proporciona.

11.1. Uso de Specialization

Specialization es una función específica para Web Beans sencillos y empresariales. Para hacer uso de Specialization, la Web Bean de prioridad más alta debe:

- ser una subclase directa del Web Bean que omite y
- ser un Web Bean sencillo si el Web Bean que omite es un Web Bean sencillo o un Web Bean empresarial si el Web Bean que omite es un Web Bean empresarial y
- estar anotado `@Specializes`.

```
@Stateless @Staging @Specializes
public class StagingCreditCardPaymentProcessor
      extends CreditCardPaymentProcessor {
   ...
}
```

Decimos que el Web Bean de prioridad más alta*especializa* su superclase.

11.2. Ventajas de Specialization

Cuando se utiliza Specialization:

- los tipos de enlace de la superclase son heredados automáticamente por el Web Bean anotado `@Specializes`, y
- el nombre de Web Bean de la superclase es heredado automáticamente por el Web Bean anotado `@Specializes`, y
- los métodos de productor, métodos desechables y métodos de observador declarados por la superclase son llamados por una instancia del Web Bean anotado `@Specializes`.

En nuestro ejemplo, el tipo de enlace `@CreditCard` de `CreditCardPaymentProcessor` es heredado por `StagingCreditCardPaymentProcessor`.

Además, el administrador de Web Bean confirmará que:

- todos los tipos de API de la superclase son tipos de API del Web Bean anotado `@Specializes` (todas las interfaces locales del bean empresarial de superclase también son interfaces locales de la subclase),
- el tipo de despliegue del Web Bean anotado `@Specializes` tiene una prioridad más alta que el tipo de despliegue de la superclase, y
- no hay ningún otro Web Bean habilitado que también especialice la superclase.

Si se violan algunas condiciones, el administrador de Web Bean produce una excepción en el momento de la inicialización.

Por lo tanto, podemos estar seguros que la superclase *nunca* será llamada en ningún despliegue del sistema donde el Web Bean anotado `@Specializes` esté desplegado y habilitado.

Definición de Web Beans utilizando XML

Hasta ahora hemos visto varios ejemplos de Web Beans declarados mediante anotaciones. No obstante, hay un par de ocasiones en que no podemos utilizar anotaciones para definir el Web Bean.

- cuando la clase de implementación viene de alguna biblioteca preexistente, o
- cuando debe haber múltiples Web Beans con la misma clase de implementación.

En estos casos, Web Beans nos ofrece dos opciones:

- escribir un método de productor, o
- declarar el Web Bean utilizando XML

Muchos marcos utilizan XML para proporcionar metadatos relacionados con clases de Java. Sin embargo, Web Beans usa un método muy diferente para especificar los nombres de clases de Java, campos o métodos para la mayoría de otros marcos. En lugar de escribir nombres de clase y miembro como valores de cadena de elementos XML y atributos, Web Beans le permite utilizar el nombre de clase o miembro como el nombre del elemento XML.

La ventaja de este enfoque es que se puede escribir un esquema de XML que evita errores de ortografía en su documento de XML. Es incluso posible que una herramienta genere automáticamente el esquema XML desde el código de Java compilado. O, un entorno de desarrollo integrado podría realizar la misma validación sin la necesidad de una etapa de generación intermedia explícita.

12.1. Declaración de clases de Web Bean

Para cada paquete de Java, Web Beans define el espacio de nombre de XML correspondiente. El espacio de nombre se forma añadiendo `urn:java:` al nombre de paquete de Java. Para el paquete `com.mydomain.myapp`, el espacio de nombre de XML es `urn:java:com.mydomain.myapp`.

Los tipos Java pertenecientes a un paquete se conocen por un elemento XML en el espacio de nombre correspondiente al paquete. El nombre del elemento es el nombre del tipo de Java y los métodos del tipo están especificados por elementos secundarios en el mismo espacio de nombre. Si el tipo es una anotación, los miembros son especificados por atributos del elemento.

Por ejemplo, el elemento `<util:Date/>` en el siguiente fragmento XML se refiere a la clase `java.util.Date`:

```
<WebBeans xmlns="urn:java:javax.webbeans"
```

```
        xmlns:util="urn:java:java.util">

   <util:Date/>

</WebBeans
>
```

¡Y este es todo el código que necesitamos para declarar que `Date` es un Web Bean sencillo! Una instancia de `Date` puede ahora ser inyectada por cualquier otro Web Bean:

```
@Current Date date
```

12.2. Declaración de metadatos de Web Bean

Podemos declarar el ámbito, el tipo de despliegue y los tipos de enlace de interceptor mediante elementos directos secundarios de la declaración del Web Bean:

```
<myapp:ShoppingCart>
   <SessionScoped/>
   <myfwk:Transactional requiresNew="true"/>
   <myfwk:Secure/>
</myapp:ShoppingCart
>
```

Utilizamos exactamente el mismo método para especificar nombres y tipo de enlace:

```
<util:Date>
   <Named
>currentTime</Named>
</util:Date>

<util:Date>
   <SessionScoped/>
   <myapp:Login/>
   <Named
>loginTime</Named>
</util:Date>

<util:Date>
   <ApplicationScoped/>
```

```
   <myapp:SystemStart/>
   <Named
>systemStartTime</Named>
</util:Date
>
```

Donde `@Login` y `@SystemStart` son tipos de anotaciones de enlace.

```
@Current Date currentTime;
@Login Date loginTime;
@SystemStart Date systemStartTime;
```

Como es usual, un Web Bean puede soportar múltiples tipos de enlace:

```
<myapp:AsynchronousChequePaymentProcessor>
   <myapp:PayByCheque/>
   <myapp:Asynchronous/>
</myapp:AsynchronousChequePaymentProcessor
>
```

Los interceptores y decoradores son sólo Web Beans sencillos, por consiguiente, pueden ser declarados como cualquier otro Web Bean sencillo:

```
<myfwk:TransactionInterceptor>
   <Interceptor/>
   <myfwk:Transactional/>
</myfwk:TransactionInterceptor
>
```

12.3. Declaración de miembros de Web Bean

¡TODO!

12.4. Declaración de Web Beans en línea

Web Beans nos permite definir un Web Bean en el punto de inyección. Por ejemplo:

```
<myapp:System>
   <ApplicationScoped/>
```

```
   <myapp:admin>
      <myapp:Name>
         <myapp:firstname
>Gavin</myapp:firstname>
         <myapp:lastname
>King</myapp:lastname>
         <myapp:email
>gavin@hibernate.org</myapp:email>
      </myapp:Name>
   </myapp:admin>
</myapp:System
>
```

El elemento `<Name>` declara un Web Bean sencillo de ámbito `@Dependent` y clase `Name`, con una serie de valores de campo iniciales. Este Web Bean tiene un enlace de contenedor-generado y es, por lo tanto, inyectable únicamente en el punto de inyección en el cual es declarado.

Esta función simple pero poderosa permite que el formato XML de Web Beans pueda utilizarse para especificar gráficos completos de objetos Java. No es del todo una solución para enlazar datos, pero ¡está cerca!

12.5. Uso de un esquema

Si deseamos que nuestro formato de documento XML sea creado por personas que no son desarrolladores de Java, o que no tienen acceso a nuestro código, necesitamos proporcionar un esquema. No hay nada específico de Web Beans sobre escribir o utilizar el esquema.

```
<WebBeans xmlns="urn:java:javax.webbeans"
      xmlns:myapp="urn:java:com.mydomain.myapp"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:java:javax.webbeans http://java.sun.com/jee/web-beans-1.0.xsd
                        urn:java:com.mydomain.myapp http://mydomain.com/xsd/myapp-1.2.xsd">

   <myapp:System>
      ...
   </myapp:System>

</WebBeans
>
```

La escritura de un esquema XML es bastante tediosa. Por lo tanto, el proyecto de la IR de Web Beans proporcionará una herramienta que genere automáticamente el esquema de XML desde el código de Java compilado.

Parte IV. Web Beans en el ecosistema de Java EE

El tercer tema de Web Beans es la *integración*. Web Beans fue diseñado para trabajar junto con otras tecnologías, ayudando al desarrollador de la aplicación a encajar en otras tecnologías.Web Beans es una tecnología abierta. Forma parte de un ecosistema de Java EE y es por si mismo una base para un nuevo ecosistema de extensiones portátiles e integración con marcos y tecnologías existentes.

Ya hemos visto cómo Web Beans ayuda a integrar EJB y JSF, permitiendo a los EJB enlazarse directamente a páginas JSF. Esto es apenas el comienzo. Web Beans ofrece el mismo potencial a otras tecnologías, tales como motores de administración de proceso de negocios, otros marcos de red y modelos de componentes de terceras partes. La plataforma de Java EE nunca podrá estandarizar todas las tecnologías interesantes utilizadas en el mundo del desarrollo de la aplicación Java, pero Web Beans facilita el uso de las tecnologías que aún no hacen parte completamente de la plataforma dentro del entorno Java EE.

Ya estamos a punto de ver cómo aprovechar completamente la plataforma de Java EE en una aplicación que utiliza Web Beans. También veremos brevemente una serie de SPI provistas para soportar extensiones a Web Beans. Puede que nunca las tenga que utilizar directamente, pero es conveniente saber que están allí si se necesitan. Lo más importante, es que podrá aprovecharlas indirectamente cada vez que utilice una extensión de terceras partes.

Integración Java EE

Los Web Beans están totalmente integrados en un entorno de Java EE. Los Web Beans tienen acceso a recursos de Java EE y a contextos persistentes de JPA. Se pueden ser utilizar en expresiones Unificadas EL en páginas JSF y JSP. Pueden ser inyectados en algunos objetos, tales como Servlets y Message Driven Beans, los cuales no son Web Beans.

13.1. Inyección de recursos de Java EE en un Web Bean

Todos los Web Beans sencillos y empresariales pueden aprovechar la inyección de dependencia de Java EE utilizando`@Resource`, `@EJB` y `@PersistenceContext`. Ya hemos visto algunos ejemplos de esto, aunque no prestamos mucha atención en el momento.

```
@Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

```
@SessionScoped
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   ...

}
```

Los `@PostConstruct` de Java EE y las llamadas de `@PreDestroy` también son compatibles con todos los Web Beans sencillos y empresariales. El método `@PostConstruct` es llamado después de realizar *toda* la inyección.

Hay una restricción para tener en cuenta aquí: `@PersistenceContext(tipo=EXTENDIDO)` no es compatible con Web Beans sencillos.

13.2. Llamando a Web Bean desde un Servlet

Es fácil utilizar un Web Bean desde un Servlet en Java EE 6. Simplemente inyecte el Web Bean mediante campo de Web Beans o Inyección de método inicializador.

```
public class Login extends HttpServlet {

   @Current Credentials credentials;
   @Current Login login;

   @Override
   public void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      credentials.setUsername( request.getAttribute("username") ):
      credentials.setPassword( request.getAttribute("password") ):
      login.login();
      if ( login.isLoggedIn() ) {
         response.sendRedirect("/home.jsp");
      }
      else {
         response.sendRedirect("/loginError.jsp");
      }
   }

}
```

El cliente proxy de Web Beans cuida las invocaciones del método de enrutamiento desde el Servlet a las instancias correctas de `Credenciales` e `Inicio de sesión` para la petición y sesión HTTP actuales.

13.3. Llamada a un Web Bean desde un Message-Driven Bean

La inyección de Web Beans se aplica a todos los EJB, incluso cuando no están bajo el control del administrador de Web Bean (si fueron obtenidos por el JNDI o inyección utilizando `@EJB`, por ejemplo). En particular, se puede utilizar inyección de Web Beans en Message-Driven Beans que no sean considerados Web Beans porque no se puede inyectarlos.

Se pueden incluso utilizar enlaces de interceptor de Web Beans para Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
```

```
   @Current Inventory inventory;
   @PersistenceContext EntityManager em;

   public void onMessage(Message message) {
      ...
   }

}
```

Así, la recepción de mensajes es superfácil en un entorno de Web Beans. No obstante, tenga en cuenta que no hay sesión o contexto de conversación disponible cuando se envía un mensaje a un Message-Driven Bean. Sólo los Web Beans `@RequestScoped` y `@ApplicationScoped` Web Beans están disponibles.

También es fácil enviar mensajes mediante Web Beans.

13.4. endpoints JMS

Enviar mensajes mediante JMS puede ser bastante complejo, debido al número de objetos diferentes que se tienen que manejar. Para colas tenemos `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` y `QueueSender`. Para temas tenemos `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` y `TopicPublisher`. Cada uno de estos objetos tiene su propio ciclo de vida y modelo de hilos de los cuales tenemos que preocuparnos.

Los Web Beans se encargan de eso por nosotros. Todo lo que se necesita es reportar la cola o tópico en `web-beans.xml`, especificando un tipo de enlace y conexión de fábrica.

```
<Queue>
   <destination
>java:comp/env/jms/OrderQueue</destination>
   <connectionFactory
>java:comp/env/jms/QueueConnectionFactory</connectionFactory>
   <myapp:OrderProcessor/>
</Queue
>
```

```
<Topic>
   <destination
>java:comp/env/jms/StockPrices</destination>
   <connectionFactory
>java:comp/env/jms/TopicConnectionFactory</connectionFactory>
```

```
    <myapp:StockPrices/>
</Topic
>
```

Ahora podemos inyectar `Queue`, `QueueConnection`, `QueueSession` o `QueueSender` para una cola, o `Topic`, `TopicConnection`, `TopicSession` o`TopicPublisher` para un tema.

```
@OrderProcessor QueueSender orderSender;
@OrderProcessor QueueSession orderSession;

public void sendMessage() {
    MapMessage msg = orderSession.createMapMessage();
    ...
    orderSender.send(msg);
}
```

```
@StockPrices TopicPublisher pricePublisher;
@StockPrices TopicSession priceSession;

public void sendMessage(String price) {
    pricePublisher.send( priceSession.createTextMessage(price) );
}
```

El ciclo de vida de objetos JMS inyectados es controlado por el administrador deWeb Bean.

13.5. Empaquetamiento y despliegue.

Web Beans no define ningún despliegue especial de archivo. Se puede empaquetar Web Beans en JAR, EJB-JAR o WAR # cualquier ubicación de despliegue en la aplicación classpath. No obstante, cada archivo que contiene Web Beans debe incluir un archivo llamado `web-beans.xml` en `META-INF` o en el directorio `WEB-INF`. El archivo puede estar vacío. Los Web Beans desplegados en archivos que no tienen un archivo `web-beans.xml` no estarán disponibles para uso en la aplicación.

Para ejecución Java SE, los Web Beans pueden ser desplegados en cualquier lugar en el que los EJB se puedan implementar para ejecución por el contenedor Lite EJB incorporable. De nuevo, cada lugar debe contener un archivo `web-beans.xml`.

Extensión de Web Beans

Web Beans pretende ser una plataforma para marcos, extensiones e integración con otras tecnologías. Por lo tanto, Web Beans expone una serie de SPI para el uso de desarrolladores de extensiones portátiles para Web Beans. Por ejemplo, las siguientes clases de extensiones fueron previstas por los diseñadores de Web Beans:

- integración con motores de Gestión de Proceso de Negocios,
- integración con marcos de terceras partes tales como Spring, Seam, GWT o Wicket, y
- nueva tecnología basada en el modelo de programación de Web Beans.

El centro nervioso para extender Web Beans es el objeto `Manager`.

14.1. El objeto `Manager`

La interfaz `Manager` nos permite registrar y obtener Web Beans, interceptores, decoradores, observadores y contextos en forma programada.

```
public interface Manager
{

  public <T
> Set<Bean<T
>
> resolveByType(Class<T
> type, Annotation... bindings);

  public <T
> Set<Bean<T
>
> resolveByType(TypeLiteral<T
> apiType,
      Annotation... bindings);

  public <T
> T getInstanceByType(Class<T
> type, Annotation... bindings);

  public <T
> T getInstanceByType(TypeLiteral<T
> type,
      Annotation... bindings);
```

```
   public Set<Bean<?>
> resolveByName(String name);

   public Object getInstanceByName(String name);

   public <T
> T getInstance(Bean<T
> bean);

   public void fireEvent(Object event, Annotation... bindings);

   public Context getContext(Class<? extends Annotation
> scopeType);

   public Manager addContext(Context context);

   public Manager addBean(Bean<?> bean);

   public Manager addInterceptor(Interceptor interceptor);

   public Manager addDecorator(Decorator decorator);

   public <T
> Manager addObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager addObserver(Observer<T
> observer, TypeLiteral<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer,
         TypeLiteral<T
```

```
> eventType, Annotation... bindings);

   public <T
> Set<Observer<T
>
> resolveObservers(T event, Annotation... bindings);

   public List<Interceptor
> resolveInterceptors(InterceptionType type,
         Annotation... interceptorBindings);

   public List<Decorator
> resolveDecorators(Set<Class<?>
> types,
         Annotation... bindings);

}
```

Podemos obtener una instancia de `Manager` vía inyección:

```
@Current Manager manager
```

14.2. La clase `Bean`

Instancias de clase abstracta `Bean` representan Web Beans. Hay una instancia de `Bean` registrada con el objeto `Manager` para cada Web Bean en la aplicación.

```
public abstract class Bean<T> {

   private final Manager manager;

   protected Bean(Manager manager) {
      this.manager=manager;
   }

   protected Manager getManager() {
      return manager;
   }

   public abstract Set<Class> getTypes();
   public abstract Set<Annotation> getBindingTypes();
   public abstract Class<? extends Annotation> getScopeType();
```

```
   public abstract Class<? extends Annotation> getDeploymentType();
   public abstract String getName();

   public abstract boolean isSerializable();
   public abstract boolean isNullable();

   public abstract T create();
   public abstract void destroy(T instance);

}
```

Es posible extender la clase `Bean` y registrar instancias llamando a `Manager.addBean()` para proveer soporte a nuevas clases de Web Beans, además de los definidos por la especificación Web Beans (Web Beans sencillos y empresariales, métodos de productor y endpoints JMS). Por ejemplo, podríamos utilizar la clase `Bean` para permitir que los objetos sean administrados por otro marco que se inyecta en Web Beans.

Hay dos subclases de `Bean` definidas por la especificación de Web Beans: `Interceptor` y `Decorador`.

14.3. La interfaz `Contexto`

La interfaz `Contexto` soporta la adición de nuevos ámbitos a Web Beans, o extensión de los ámbitos incorporados a nuevos entornos.

```
public interface Context {

   public Class<? extends Annotation> getScopeType();

   public <T> T get(Bean<T> bean, boolean create);

   boolean isActive();

}
```

Por ejemplo, podríamos implementar `Contexto` para agregar un ámbito de proceso de negocio a Web Beans, o agregar soporte para el ámbito de conversación a una aplicación que utiliza Wicket.

Siguientes pasos

Debido a que Web Beans es tan reciente, aún no hay mucha información en línea disponible.

Claro está que la especificación de Web Beans es la mejor fuente de información sobre Web Beans. La especificación tiene cerca de 100 páginas, apenas dos veces el tamaño de este artículo y casi tan legible. No obstante, cubre muchos detalles que hemos omitido. La especificación está disponible en `http://jcp.org/en/jsr/detail?id=299`.

La implementación de Referencia (IR) de Web Beans se desarrolla en `http://seamframework.org/WebBeans`. El equipo de desarrollo de la IR y la especificación de Web Beans lideran el Blog en `http://in.relation.to`. Este artículo se base en una serie de entradas de Blog publicadas allí.

Parte V. Web Beans Reference

Web Beans is the reference implementation of JSR-299, and is used by JBoss AS and Glassfish to provide JSR-299 services for Java Enterprise Edition applications. Web Beans also goes beyond the environments and APIs defined by the JSR-299 specification and provides support for a number of other environments (such as a servlet container such as Tomcat, or Java SE) and additional APIs and modules (such as logging, XSD generation for the JSR-299 XML deployment descriptors).

If you want to get started quickly using Web Beans with JBoss AS or Tomcat and experiment with one of the examples, take a look at *Capítulo 3, Getting started with Web Beans, the Reference Implementation of JSR-299*. Otherwise read on for a exhaustive discussion of using Web Beans in all the environments and application servers it supports, as well the Web Beans extensions.

Application Servers and environments supported by Web Beans

16.1. Using Web Beans with JBoss AS

No special configuration of your application, beyond adding either `META-INF/beans.xml` or `WEB-INF/beans.xml` is needed.

If you are using JBoss AS 5.0.1.GA then you'll need to install Web Beans as an extra. First we need to tell Web Beans where JBoss is located. Edit `jboss-as/build.properties` and set the `jboss.home` property. For example:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

Now we can install Web Beans:

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Nota

A new deployer, `webbeans.deployer` is added to JBoss AS. This adds supports for JSR-299 deployments to JBoss AS, and allows Web Beans to query the EJB3 container and discover which EJBs are installed in your application.

Web Beans is built into all releases of JBoss AS from 5.1 onwards.

16.2. Glassfish

TODO

16.3. Servlet Containers (such as Tomcat or Jetty)

Web Beans can be used in any Servlet container such as Tomcat 6.0 or Jetty 6.1.

Nota

Web Beans doesn't support deploying session beans, injection using `@EJB`, or `@PersistenceContext` or using transactional events in Servlet containers.

Web Beans should be used as a web application library in a servlet container. You should place `webbeans-servlet.jar` in `WEB-INF/lib`. `webbeans-serv;et.jar` is an "uber-jar" provided for your convenience. Instead, you could use its component jars:

- `jsr299-api.jar`
- `webbeans-api.jar`
- `webbeans-spi.jar`
- `webbeans-core.jar`
- `webbeans-logging.jar`
- `webbeans-servlet-int.jar`
- `javassist.jar`
- `dom4j.jar`

You also need to explicitly specify the servlet listener (used to boot Web Beans, and control its interaction with requests) in `web.xml`:

```
<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

16.3.1. Tomcat

Tomcat has a read-only JNDI, so Web Beans can't automatically bind the Manager. To bind the Manager into JNDI, you should add the following to your `META-INF/context.xml`:

```
<Resource name="app/Manager"
        auth="Container"
        type="javax.inject.manager.Manager"
        factory="org.jboss.webbeans.resources.ManagerObjectFactory"/>
```

and make it available to your deployment by adding this to `web.xml`:

```
<resource-env-ref>
  <resource-env-ref-name>
    app/Manager
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.inject.manager.Manager
  </resource-env-ref-type>
</resource-env-ref>
```

Tomcat only allows you to bind entries to `java:comp/env`, so the Manager will be available at `java:comp/env/app/Manager`

Web Beans also supports Servlet injection in Tomcat. To enable this, place the `webbeans-tomcat-support.jar` in `$TOMCAT_HOME/lib`, and add the following to your `META-INF/context.xml`:

```
<Listener className="org.jboss.webbeans.environment.tomcat.WebBeansLifecycleListener" />
```

16.4. Java SE

Apart from improved integration of the Enterprise Java stack, Web Beans also provides a state of the art typesafe, stateful dependency injection framework. This is useful in a wide range of application types, enterprise or otherwise. To facilitate this, Web Beans provides a simple means for executing in the Java Standard Edition environment independently of any Enterprise Edition features.

When executing in the SE environment the following features of Web Beans are available:

- `Simple Web Beans (POJOs)`
- `Typesafe Dependency Injection`
- `Application and Dependent Contexts`
- `Binding Types`
- `Stereotypes`
- `Typesafe Event Model`

16.4.1. Web Beans SE Module

To make life easy for developers Web Beans provides a special module with a main method which will boot the Web Beans manager, automatically registering all simple Web Beans found on the classpath. This eliminates the need for application developers to write any bootstrapping

code. The entry point for a Web Beans SE applications is a simple Web Bean which observes the standard `@Deployed Manager` event. The command line paramters can be injected using either of the following:

```
@Parameters List<String> params;
@Parameters String[] paramsArray; // useful for compatability with existing classes
```

Here's an example of a simple Web Beans SE application:

```
@ApplicationScoped
public class HelloWorld
{
   @Parameters List<String> parameters;

   public void printHello( @Observes @Deployed Manager manager )
   {
      System.out.println( "Hello " + parameters.get(0) );
   }
}
```

Web Beans SE applications are started by running the following main method.

```
java org.jboss.webbeans.environments.se.StartMain <args>
```

If you need to do any custom initialization of the Web Beans manager, for example registering custom contexts or initializing resources for your beans you can do so in response to the `@Initialized Manager` event. The following example registers a custom context:

```
public class PerformSetup
{

   public void setup( @Observes @Initialized Manager manager )
   {
      manager.addContext( ThreadContext.INSTANCE );
   }
}
```

Nota

The command line parameters do not become available for injection until the `@Deployed Manager` event is fired. If you need access to the parameters during initialization you can do so via the `public static String getParameters()` method in `StartMain`.

JSR-299 extensions available as part of Web Beans

Importante

These modules are usable on any JSR-299 implementation, not just Web Beans!

17.1. Web Beans Logger

Adding logging to your application is now even easier with simple injection of a logger object into any JSR-299 bean. Simply annotate a org.jboss.webbeans.log.Log type member with *@Logger* and an appropriate logger object will be injected into any instance of the bean.

```
public class Checkout {
   import org.jboss.webbeans.annotation.Logger;
   import org.jboss.webbeans.log.Log;

   @Logger
   private Log log;

   void invoiceItems() {
      ShoppingCart cart;
      . . .
      log.debug("Items invoiced for {0}", cart);
   }

}
```

The example shows how objects can be interpolated into a message. This interpolation is done using *java.text.MessageFormat*, so see the JavaDoc for that class for more details. In this case, the ShoppingCart should have implemented the *toString()* method to produce a human readable value that is meaningful in messages. Normally, this call would have involved evaluating cart.toString() with String concatenation to produce a single String argument. Thus it was necessary to surround the call with an if-statement using the condition *log.isDebugEnabled()* to avoid the expensive String concatenation if the message was not actually going to be used. However, when using @Logger injected logging, the conditional test can be left out since the object arguments are not evaluated unless the message is going to be logged.

Nota

You can add the Web Beans Logger to your project by including webbeans-logger.jar and webbeans-logging.jar to your project. Alternatively, express a dependency on the `org.jboss.webbeans:webbeans-logger` Maven artifact.

If you are using Web Beans as your JSR-299 implementation, there is no need to include `webbeans-logging.jar` as it's already included.

Alternative view layers

18.1. Using Web Beans with Wicket

18.1.1. The `WebApplication` class

Each wicket application must have a `WebApplication` subclass; Web Beans provides, for your utility, a subclass of this which sets up the Wicket/JSR-299 integration. You should subclass `org.jboss.webbeans.wicket.WebBeansApplication`.

Nota

If you would prefer not to subclass `WebBeansApplication`, you can manually add a (small!) number of overrides and listeners to your own `WebApplication` subclass. The javadocs of `WebBeansApplication`detail this.

For example:

```
public class SampleApplication extends WebBeansApplication {
   @Override
   public Class getHomePage() {
      return HomePage.class;
   }
}
```

18.1.2. Conversations with Wicket

The conversation scope can be used in Web Beans with the Apache Wicket web framework, through the `webbeans-wicket` module. This module takes care of:

- Setting up the conversation context at the beginning of a Wicket request, and tearing it down afterwards
- Storing the id of any long-running conversation in Wicket's metadata when the page response is complete
- Activating the correct long-running conversation based upon which page is being accessed
- Propagating the conversation context for any long-running conversation to new pages

18.1.2.1. Starting and stopping conversations in Wicket

As JSF applications, a conversation *always* exists for any request, but its lifetime is only that of the current request unless it is marked as *long-running*. For Wicket applications this is

accomplished as in JSF applications, by injecting the `@Current Conversation` and then invoking `conversation.begin()`. Likewise, conversations are ended with `conversation.end()`

18.1.2.2. Long running conversation propagation in Wicket

When a conversation is marked as long-running, the id of that conversation will be stored in Wicket's metadata for the current page. If a new page is created and set as the response target through `setResponsePage`, this new page will also participate in this conversation. This occurs for both directly instantiated pages (`setResponsePage(new OtherPage())`), as well as for bookmarkable pages created with `setResponsePage(OtherPage.class)` where `OtherPage.class` is mounted as bookmarkable from your `WebApplication` subclass (or through annotations). In the latter case, because the new page instance is not created until after a redirect, the conversation id will be propagated through a request parameter, and then stored in page metadata after the redirect.

Apéndice A. Integrating Web Beans into other environments

Currently Web Beans only runs in JBoss AS 5; integrating the RI into other EE environments (for example another application server like Glassfish), into a servlet container (like Tomcat), or with an Embedded EJB3.1 implementation is fairly easy. In this Appendix we will briefly discuss the steps needed.

A.1. The Web Beans SPI

The Web Beans SPI is located in the `webbeans-spi` module, and packaged as `webbeans-spi.jar`. Some SPIs are optional, if you need to override the default behavior, others are required.

All interfaces in the SPI support the decorator pattern and provide a `Forwarding` class located in the `helpers` sub package. Additional, commonly used, utility classes, and standard implementations are also located in the `helpers` sub package.

A.1.1. Web Bean Discovery

```
/**
   * Gets list of all classes in classpath archives with META-INF/beans.xml (or
   * for WARs WEB-INF/beans.xml) files
   *
   * @return An iterable over the classes
   */
  public Iterable<Class<?>> discoverWebBeanClasses();

  /**
   * Gets a list of all deployment descriptors in the app classpath
   *
   * @return An iterable over the beans.xml files
   */
  public Iterable<URL> discoverWebBeansXml();
```

The discovery of Web Bean classes and `beans.xml` files is self-explanatory (the algorithm is described in Section 11.1 of the JSR-299 specification, and isn't repeated here).

A.1.2. EJB services

Nota

Web Beans will run without an EJB container; in this case you don't need to implement the EJB SPI.

Web Beans also delegates EJB3 bean discovery to the container so that it doesn't have to scan for EJB3 annotations or parse `ejb-jar.xml`. For each EJB in the application an EJBDescriptor should be discovered:

```
public interface EjbDescriptor<T>
{

  /**
   * Gets the EJB type
   *
   * @return The EJB Bean class
   */
  public Class<T> getType();

  /**
   * Gets the local business interfaces of the EJB
   *
   * @return An iterator over the local business interfaces
   */
  public Iterable<BusinessInterfaceDescriptor<?>> getLocalBusinessInterfaces();

  /**
   * Gets the remote business interfaces of the EJB
   *
   * @return An iterator over the remote business interfaces
   */
  public Iterable<BusinessInterfaceDescriptor<?>> getRemoteBusinessInterfaces();

  /**
   * Get the remove methods of the EJB
   *
   * @return An iterator over the remove methods
   */
  public Iterable<Method> getRemoveMethods();

  /**
```

```
     * Indicates if the bean is stateless
     *
     * @return True if stateless, false otherwise
     */
    public boolean isStateless();

    /**
     * Indicates if the bean is a EJB 3.1 Singleton
     *
     * @return True if the bean is a singleton, false otherwise
     */
    public boolean isSingleton();

    /**
     * Indicates if the EJB is stateful
     *
     * @return True if the bean is stateful, false otherwise
     */
    public boolean isStateful();

    /**
     * Indicates if the EJB is and MDB
     *
     * @return True if the bean is an MDB, false otherwise
     */
    public boolean isMessageDriven();

    /**
     * Gets the EJB name
     *
     * @return The name
     */
    public String getEjbName();
```

El `EjbDescriptor` es bastante auto explicativo y debería devolver los metadatos pertinentes como se define en la especificación de EJB. Además de estas dos interfaces, está `BusinessInterfaceDescriptor`, la cual representa una interfaz de negocios local (encapsulando la clase de interfaz y el nombre de jndi a la búsqueda de una instancia del EJB).

The resolution of `@EJB` (for injection into simple beans), the resolution of local EJBs (for backing session beans) and remote EJBs (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `org.jboss.webbeans.ejb.spi.EjbServices` which provides these operations. For resolving the `@EJB` injection point, Web Beans will provide the

`InjectionPoint`; for resolving local EJBs, the `EjbDescriptor` will be provided, and for remote EJBs the `jndiName`, `mappedName`, or `ejbLink` will be provided.

When resolving local EJBs (used to back session beans) a wrapper (`SessionObjectReference`) around the EJB reference is returned. This wrapper allows Web Beans to request a reference that implements the given business interface, and, in the case of SFSBs, request the removal of the EJB from the container.

A.1.3. JPA services

Just as EJB resolution is delegated to the container, resolution of `@PersistenceContext` for injection into simple beans (with the `InjectionPoint` provided), and resolution of persistence contexts and persistence units (with the `unitName` provided) for injection as a Java EE resource is delegated to the container.

To allow JPA integration, the `JpaServices` interface should be implemented.

Web Beans also needs to know what entities are in a deployment (so that they aren't managed by Web Beans). An implementation that detects entities through `@Entity` and `orm.xml` is provided by default. If you want to provide support for a entities defined by a JPA provider (such as Hibernate's `.hbm.xml` you can wrap or replace the default implementation.

```
EntityDiscovery delegate = bootstrap.getServices().get(EntityDiscovery.class);
```

A.1.4. Transaction Services

Web Beans must delegate JTA activities to the container. The SPI provides a couple hooks to easily achieve this with the `TransactionServices` interface.

```
public interface TransactionServices
{
  /**
   * Possible status conditions for a transaction. This can be used by SPI
   * providers to keep track for which status an observer is used.
   */
  public static enum Status
  {
    ALL, SUCCESS, FAILURE
  }

  /**
   * Registers a synchronization object with the currently executing
   * transaction.
   *
```

```
    * @see javax.transaction.Synchronization
    * @param synchronizedObserver
    */
   public void registerSynchronization(Synchronization synchronizedObserver);

   /**
    * Queries the status of the current execution to see if a transaction is
    * currently active.
    *
    * @return true if a transaction is active
    */
   public boolean isTransactionActive();
}
```

The enumeration `Status` is a convenience for implementors to be able to keep track of whether a synchronization is supposed to notify an observer only when the transaction is successful, or after a failure, or regardless of the status of the transaction.

Any `javax.transaction.Synchronization` implementation may be passed to the `registerSynchronization()` method and the SPI implementation should immediately register the synchronization with the JTA transaction manager used for the EJBs.

To make it easier to determine whether or not a transaction is currently active for the requesting thread, the `isTransactionActive()` method can be used. The SPI implementation should query the same JTA transaction manager used for the EJBs.

A.1.5. JMS services

A number of JMS operations are not container specific, and so should be provided via the SPI `JmsServices`. JMS does not specify how to obtain a `ConnectionFactory` so the SPI provides a method which should be used to look up a factory. Web Beans also delegates `Destination` lookup to the container via the SPI.

A.1.6. Resource Services

The resolution of `@Resource` (for injection into simple beans) and the resolution of resources (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `ResourceServices` which provides these operations. For resolving the `@Resource` injection, Web Beans will provide the `InjectionPoint`; and for Java EE resources, the `jndiName` or `mappedName` will be provided.

A.1.7. Web Services

The resolution of web service references (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `WebServices` whichprovides this operation. For resolving the Java EE resource, the `jndiName` or `mappedName` will be provided.

A.1.8. The bean store

Web Beans uses a map like structure to store bean instances - `org.jboss.webbeans.context.api.BeanStore`. You may find `org.jboss.webbeans.context.api.helpers.ConcurrentHashMapBeanStore` useful.

A.1.9. The application context

Web Beans expects the Application Server or other container to provide the storage for each application's context. The `org.jboss.webbeans.context.api.BeanStore` should be implemented to provide an application scoped storage.

A.1.10. Bootstrap and shutdown

The `org.jboss.webbeans.bootstrap.api.Bootstrap` interface defines the bootstrap for Web Beans. To boot Web Beans, you must obtain an instance of `org.jboss.webbeans.bootstrap.WebBeansBootstrap` (which implements `Boostrap`), tell it about the SPIs in use, and then request the container start.

The bootstrap is split into phases, bootstrap initialization and boot and shutdown. Initialization will create a manager, and add the standard (specification defined) contexts. Bootstrap will discover EJBs, classes and XML; add beans defined using annotations; add beans defined using XML; and validate all beans.

The bootstrap supports multiple environments. An environment is defined by an implementation of the `Environment` interface. A number of standard envrionments are built in as the enumeration `Environments`. Different environments require different services to be present (for example servlet doesn't require transaction, EJB or JPA services). By default an EE environment is assumed, but you can adjust the environment by calling `bootstrap.setEnvironment()`.

Web Beans uses a generic-typed service registry to allow services to be registered. All services implement the `Service` interface. The service registry allows services to be added and retrieved.

To initialize the bootstrap you call `Bootstrap.initialize()`. Before calling `initialize()`, you must register any services required by your environment. You can do this by calling `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. You must also provide the application context bean store.

Having called `initialize()`, the `Manager` can be obtained by calling `Bootstrap.getManager()`.

To boot the container you call `Bootstrap.boot()`.

To shutdown the container you call `Bootstrap.shutdown()` or `webBeansManager.shutdown()`. This allows the container to perform any cleanup operations needed.

A.1.11. JNDI

Web Beans delegates all JNDI operations to the container through the SPI.

Nota

A number of the SPI interface require JNDI lookup, and the class `AbstractResourceServices` provides JNDI/Java EE spec compliant lookup methods.

A.1.12. Carga de recurso

Web Beans needs to load classes and resources from the classpath at various times. By default, they are loaded from the Thread Context ClassLoader if available, if not the same classloader that was used to load Web Beans, however this may not be correct for some environments. If this is case, you can implement `org.jboss.webbeans.spi.ResourceLoader`:

```
    public interface ResourceLoader {

  /**
   * Creates a class from a given FQCN
   *
   * @param name The name of the clsas
   * @return The class
   */
  public Class<?> classForName(String name);

  /**
   * Gets a resource as a URL by name
   *
   * @param name The name of the resource
   * @return An URL to the resource
   */
  public URL getResource(String name);

  /**
   * Gets resources as URLs by name
   *
   * @param name The name of the resource
   * @return An iterable reference to the URLS
   */
  public Iterable<URL
> getResources(String name);

}
```

A.1.13. Servlet injection

Java EE / Servlet does not provide any hooks which can be used to provide injection into Servlets, so Web Beans provides an API to allow the container to request JSR-299 injection for a Servlet.

To be compliant with JSR-299, the container should request servlet injection for each newly instantiated servlet after the constructor returns and before the servlet is placed into service.

To perform injection on a servlet call `WebBeansManager.injectServlet()`. The manager can be obtained from `Bootstrap.getManager()`.

A.2. El contrato con el contenedor

Hay una serie de requisitos que la IR de Web Beans ubica en el contenedor para el funcionamiento correcto que está fuera de la implementación de las API.

Aislamiento de classloader
: Si se está integrando la IR de Web Beans dentro de un entorno que admite despliegue de varias aplicaciones, debe habilitar el aislamiento de classloader para cada aplicación de Web Beans, de forma automática o a través de la configuración del usuario.

Servlet
: Si usted está integrando el Web Beans en un entorno de Servlet debe registrar `org.jboss.webbeans.servlet.WebBeansListener` como oyente de Servlet, ya sea automáticamente o a través de la configuración de usuario, para cada aplicación Web Beans que utiliza Servlet.

JSF
: If you are integrating the Web Beans into a JSF environment you must register `org.jboss.webbeans.jsf.WebBeansPhaseListener` as a phase listener, and `org.jboss.webbeans.el.WebBeansELREsolver` as an EL resolver, either automatically, or through user configuration, for each Web Beans application which uses JSF.

  If you are integrating the Web Beans into a JSF environment you must register `org.jboss.webbeans.servlet.ConversationPropagationFilter` as a Servlet listener, either automatically, or through user configuration, for each Web Beans application which uses JSF. This filter can be registered for all Servlet deployment safely.

Nota

Web Beans only supports JSF 1.2 and above.

Intercepción de sesión de Bean

Si está integrando los Web Beans en un entorno EJB debe registrar `org.jboss.webbeans.ejb.SessionBeanInterceptor` como un interceptor EJB para todos los EJB en la aplicación, automáticamente o a través de la configuración de usuario, para cada aplicación que utilice Web Beans empresariales.

Importante

You must register the `SessionBeanInterceptor` as the inner most interceptor in the stack for all EJBs.

The `webbeans-core.jar`

If you are integrating the Web Beans into an environment that supports deployment of applications, you must insert the `webbeans-core.jar` into the applications isolated classloader. It cannot be loaded from a shared classloader.

Binding the manager in JNDI

You should bind a `Reference` to the Manager `ObjectFactory` into JNDI at `java:app/Manager`. The type should be `javax.inject.manager.Manager` and the factory class is `org.jboss.webbeans.resources.ManagerObjectFactory`