## Web Beans: Contextos Java e Injeção de Dependência

# O novo padrão para injeção de dependência e gerenciamento de estado contextual

**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**
**Italian Translation: 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

## Nota

A JSR-299 mudou recentemente seu nome de "Web Beans" para "Contextos Java e InjeÃ§Ã£o de DependÃªncia". O guia de referÃªncia ainda se refere a JSR-299 como "Web Beans" e a ImplementaÃ§Ã£o de ReferÃªncia da JSR-299 como "Web Beans RI". Outra documentaÃ§Ã£o, blogs, fÃ³runs, etc podem utilizar a nova nomenclatura, incluindo o novo nome para a ImplementaÃ§Ã£o de ReferÃªncia da JSR-299 - "Web Beans".

VocÃª tambÃ©m descobrirÃ¡ que algumas das mais recentes funcionalidades nÃ£o estÃ£o implementadas (como campos produtores, realizaÃ§Ã£o, eventos assÃ-ncronos, mapeamento XML dos recursos EE).

# Parte I. Utilizando objetos contextuais

A especificação Web Beans (JSR-299) define um conjunto de serviços para o ambiente Java EE, o que torna muito simples o desenvolvimento de aplicações. Web Beans adiciona um avançado ciclo de vida e um modelo interativo sobre os tipos de componentes Java existentes, incluindo os JavaBeans e Enterprise Java Beans. Como complemento ao tradicional modelo de programação Java EE, a Web Beans provê:

- um ciclo de vida melhorado para componentes stateful, vinculados a *contextos* bem definidos,
- uma abordagem typesafe para *injeção de dependência*,
- interação via *notificação de eventos*, e
- uma melhor abordagem para associar *interceptadores* a componentes, juntamente com um novo tipo de interceptador, chamado de *decorador*, que é mais adequado para utilização na resolução de problemas de negócio.

Injeção de dependência, juntamente com o gerenciamento contextual do ciclo de vida, livra o usuário de uma API desconhecida de ter de perguntar e reponders às seguintes questões:

- qual o ciclo de vida desse objeto?
- quantos clientes simultâneos eu posso ter?
- é multithread?
- de onde posso obter um?
- eu preciso explicitamente destruí-lo?
- onde devo manter minha referência quando não estou usando-o diretamente?
- como posso adicionar uma camada de indireção, de modo que a implementação desse objeto possa variar em tempo de implantação?
- como compartilhar esse objeto entre outros objetos?

Um Web Bean especifica apenas o tipo e a semântica de outros Web Beans dos quais que ele dependa. Ele não precisa ser consciente do próprio ciclo de vida, implementação concreta, modelo de threading ou outro cliente de qualquer Web Bean de que ele dependa.Melhor ainda: a implementação concreta, ciclo de vida e o modelo de threading do Web Bean de que ele dependa podem variar de acordo com o cenário de implantação, sem afetar qualquer cliente.

Eventos, interceptadores e decoradores permitem o *fraco acoplamento* que é inerente nesse modelo:

- *notificadores de eventos* desacoplam os produtores de eventos dos consumidores dos eventos,
- *interceptadores* desacoplam questões técnicas da lógica de negócios, e
- *decoradores* permitem que questões de negócios sejam compartimentadas.

Mais importante, Web Beans oferece todas essas facilidades de uma maneira *typesafe*. Web Beans nunca utiliza identificadores baseados em strings para determinar o modo como os objetos se relacionam. XML continua a ser uma opção, mas raramente é utilizado. Em vez disso, Web Beans utiliza a informação de tipo que está disponível no modelo de objeto Java, juntamente com um novo padrão, chamado *anotações de binding*, para interconectar Web Beans, suas dependências, seus interceptadores e decoradores e seus consumidores de eventos.

Os serviços dos Web Beans são genéricos e aplicados aos seguintes tipo de componentes existentes no ambiente Java EE:

- todos JavaBeans,
- todos EJBs, e
- todos os Servlets.

Web Beans provê ainda pontos de integração necessários para que outros tipos de componentes definidos pelas futuras especificações Java EE ou por frameworks não-padrão possam ser transparentemente integrados com a Web Beans, tirando proveito dos serviços da Web Beans e interagindo com qualquer outro tipo de Web Bean.

A Web Beans foi influenciada por inúmeros frameworks Java existentes, incluindo Seam, Guice e Spring. Entretanto, Web Beans tem suas prórias características: mais typesafe que o Seam, mais stateful e menos centrada em XML que o Spring, mais web e capaz para aplicações corporativas que o Guice.

O mais importante: Web Beans é um padrão do JCP, que se integra transparentemente com o Java EE e com qualquer outro ambiente Java SE em que o EJB Lite embutível esteja disponível.

# Introdução a Web Beans

Então você está interessado em começar a escrever o seu primeiro Web Bean? Ou talvez você é cético, imaginando que tipos de hoops a especificação Web Beans fará com que você passe! A boa notícia é que você provavelmente já escreveu e utilizou centenas, talvez milhares de Web Beans. Você pode até não se lembrar do primeiro Web Bean que escreveu.

## 1.1. Seu primeiro Web Bean

Com certeza, com raras exceções especiais, toda classe Java com um construtor sem parâmetros é um Web Bean. Isso inclui todos os JavaBean. Além disso, todo session bean no estilo EJB 3 é um Web Bean. Claro, JavaBeans e EJBs que você tem escrito todos os dias, não tem sido capazes de aproveitar os novos serviços definidos pela especificação Web Beans, mas você será capaz de usar cada um deles como Web Beans - injetando-os em outros Web Beans, configurando-os através da Web Beans XML configuration facility, e até acrescentando interceptadores e decoradores # sem tocar o seu código existente.

Suponha que temos duas classes Java existentes, que temos utilizado por anos em várias aplicações. A primeira classe faz a divisão (parse) de uma string em uma lista de sentenças:

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

A segunda classe existente é um stateless session bean de fachada (front-end) para um sistema externo que é capaz de traduzir frases de uma língua para outra:

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

Onde `Translator` é a interface local:

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

Infelizmente, não temos uma classe pré-existente que traduz todo o texto de documentos. Então vamos escrever um Web Bean que faz este trabalho:

```
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 obter uma instância de `TextTranslator` injetando-a em um Web Bean, Servlet ou EJB:

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

Alternativamente, nós podemos obter uma instância invocando diretamente o método do gerenciador do Web Bean:

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

Mas espere: `TextTranslator` não tem um construtor sem parâmetros! É ainda um Web Bean? Bem, uma classe que não tem um construtor sem parâmetros ainda pode ser um Web Bean, se tiver um construtor anotado com `@Initializer`.

As you've guessed, the `@Initializer` annotation has something to do with dependency injection! `@Initializer` may be applied to a constructor or method of a Web Bean, and tells the Web Bean manager to call that constructor or method when instantiating the Web Bean. The Web Bean manager will inject other Web Beans to the parameters of the constructor or method.

At system initialization time, the Web Bean manager must validate that exactly one Web Bean exists which satisfies each injection point. In our example, if no implementation of `Translator` available # if the `SentenceTranslator` EJB was not deployed # the Web Bean manager would throw an `UnsatisfiedDependencyException`. If more than one implementation of `Translator` was available, the Web Bean manager would throw an `AmbiguousDependencyException`.

## 1.2. O que é um Web Bean?

Então, o que, *exatamente*, é um Web Bean?

A Web Bean is an application class that contains business logic. A Web Bean may be called directly from Java code, or it may be invoked via Unified EL. A Web Bean may access transactional resources. Dependencies between Web Beans are managed automatically by the Web Bean manager. Most Web Beans are *stateful* and *contextual*. The lifecycle of a Web Bean is always managed by the Web Bean manager.

Let's back up a second. What does it really mean to be "contextual"? Since Web Beans may be stateful, it matters *which* bean instance I have. Unlike a stateless component model (for example, stateless session beans) or a singleton component model (such as servlets, or singleton beans), different clients of a Web Bean see the Web Bean in different states. The client-visible state depends upon which instance of the Web Bean the client has a reference to.

No entanto, como o modelo stateless ou singleton, mas *ao contrário* dos stateful session beans, o cliente não controla o ciclo de vida da instância por explicitamente criar e destruí-lo. Em vez disso, o *escopo* do Web Bean determina:

- o ciclo de vida de cada instância do Web Bean e
- que os clientes compartilham uma referência a uma instância específica do Web Bean.

For a given thread in a Web Beans application, there may be an *active context* associated with the scope of the Web Bean. This context may be unique to the thread (for example, if the Web Bean is request scoped), or it may be shared with certain other threads (for example, if the Web Bean is session scoped) or even all other threads (if it is application scoped).

Os clientes (por exemplo, outros Web Beans) executam no mesmo contexto verão a mesma instância do Web Bean. Mas os clientes em um contexto diferente verão uma outra instância.

One great advantage of the contextual model is that it allows stateful Web Beans to be treated like services! The client need not concern itself with managing the lifecycle of the Web Bean it is using, *nor does it even need to know what that lifecyle is.* Web Beans interact by passing messages,

and the Web Bean implementations define the lifecycle of their own state. The Web Beans are loosely coupled because:

- ele interagem por uma API pública bem definida
- seus ciclos de vida são completamente desacoplados

We can replace one Web Bean with a different Web Bean that implements the same API and has a different lifecycle (a different scope) without affecting the other Web Bean implementation. In fact, Web Beans defines a sophisticated facility for overriding Web Bean implementations at deployment time, as we will see in *Seção 4.2, “Tipo de deploy”*.

Note que nem todos os clientes de um Web Bean são Web Beans. Outros objetos, tais como Servlets ou Message-Driven Beans # que são, por natureza, não injetável, objetos contextuais # podem também obter referências a Web Beans por injeção.

Chega de mão abanando. Mais formalmente, de acordo com a especificação:

> Um Web Bean compreende:
>
> - Um conjunto (não vazio) de tipos de API (API types)
> - Um conjunto (não vazio) de tipos de anotações de binding
> - Um escopo
> - Um tipo de publicação (deployment type)
> - Opcionalmente, um nome Web Bean
> - Um conjunto de tipos de interceptor binding
> - A implementação de Web Bean

Vamos ver o que alguns destes termos significam, para o desenvolvedor Web Bean.

### 1.2.1. Tipos de API, tipos de binding e injeção de dependências

Web Beans usually acquire references to other Web Beans via dependency injection. Any injected attribute specifies a "contract" that must be satisfied by the Web Bean to be injected. The contract is:

- um tipo de API, juntamente com
- um conjunto de tipos de binding

An API is a user-defined class or interface. (If the Web Bean is an EJB session bean, the API type is the `@Local` interface or bean-class local view). A binding type represents some client-visible semantic that is satisfied by some implementations of the API and not by others.

Binding types are represented by user-defined annotations that are themselves annotated `@BindingType`. For example, the following injection point has API type `PaymentProcessor` and binding type `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

If no binding type is explicitly specified at an injection point, the default binding type `@Current` is assumed.

For each injection point, the Web Bean manager searches for a Web Bean which satisfies the contract (implements the API, and has all the binding types), and injects that Web Bean.

The following Web Bean has the binding type `@CreditCard` and implements the API type `PaymentProcessor`. It could therefore be injected to the example injection point:

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

If a Web Bean does not explicitly specify a set of binding types, it has exactly one binding type: the default binding type `@Current`.

Web Beans defines a sophisticated but intuitive *resolution algorithm* that helps the container decide what to do if there is more than one Web Bean that satisfies a particular contract. We'll get into the details in *Capítulo 4, Injeção de Dependências*.

## 1.2.2. Tipos de publicação (deployment types)

*Deployment types* let us classify our Web Beans by deployment scenario. A deployment type is an annotation that represents a particular deployment scenario, for example `@Mock`, `@Staging` or `@AustralianTaxLaw`. We apply the annotation to Web Beans which should be deployed in that scenario. A deployment type allows a whole set of Web Beans to be conditionally deployed, with a just single line of configuration.

Many Web Beans just use the default deployment type `@Production`, in which case no deployment type need be explicitly specified. All three Web Bean in our example have the deployment type `@Production`.

In a testing environment, we might want to replace the `SentenceTranslator` Web Bean with a "mock object":

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

We would enable the deployment type `@Mock` in our testing environment, to indicate that `MockSentenceTranslator` and any other Web Bean annotated `@Mock` should be used.

Iremos falar mais sobre essa única e poderosa funcionalidade em *Seção 4.2, "Tipo de deploy"*.

### 1.2.3. Escopo

The *scope* defines the lifecycle and visibility of instances of the Web Bean. The Web Beans context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built-in to the specification, and provided by the Web Bean manager. A scope is represented by an annotation type.

Por exemplo, qualquer aplicação web pode ter Web Beans com *escopo de sessão (session scoped)*:

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

An instance of a session scoped Web Bean is bound to a user session and is shared by all requests that execute in the context of that session.

Por padrão, os Web Beans pertencem a um escopo especial chamado de *dependent pseudo-scope*. Web Beans com este escopo são objetos puramente dependentes do objeto em que são injetados, e seu ciclo de vida está atrelado ao ciclo de vida desse objeto.

Falaremos mais sobre escopos no *Capítulo 5, Escopos e contextos*.

### 1.2.4. Nomes Web Beans e Unified EL

Um Web Bean pode ter um *name*, que lhe permite ser utilizado em expressões da Unified EL. É fácil especificar o nome de um Web Bean:

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

Agora, podemos facilmente utilizar o Web Bean em qualquer página JSF ou JSP:

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

É ainda mais fácil, deixar o nome ser atribuído pelo gerenciador do Web Bean:

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

Neste caso, o nome fica `shoppingCart` # o nome da classe não qualificado (unqualified class name), com o primeiro caractere alterado para minúsculas.

### 1.2.5. Tipos de interceptor binding

Web Beans suporta a funcionalidade de interceptador (interceptor) definida pela EJB 3, não apenas para beans EJB , mas também para classes Java simples (plain Java classes). Além disso, a Web Beans oferece uma nova abordagem para vincular interceptores (binding interceptors) para beans EJB e outros Web Beans.

It remains possible to directly specify the interceptor class via use of the `@Interceptors` annotation:

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

No entanto, é mais elegante, e uma melhor prática, indirecionar o binding do interceptador através de um *interceptor binding type*:

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

We'll discuss Web Beans interceptors and decorators in *Capítulo 7, Interceptadores* and *Capítulo 8, Decoradores*.

## 1.3. Que tipos de objetos podem ser Web Beans?

We've already seen that JavaBeans, EJBs and some other Java classes can be Web Beans. But exactly what kinds of objects are Web Beans?

### 1.3.1. Web Beans Simples

A especificação de Web Beans diz que uma classe Java concreta é um Web Bean *simples* se:

- não é um componente gerenciado pelo container, como um EJB, um Servlet ou uma entidade da JPA,
- it is not a non-static static inner class,
- não é um tipo parametrizado, e
- que tem um construtor sem parâmetros, ou um construtor anotado com `@Initializer`.

Assim, quase todo JavaBean é um Web Bean simples.

Every interface implemented directly or indirectly by a simple Web Bean is an API type of the simple Web Bean. The class and its superclasses are also API types.

### 1.3.2. Web Beans corporativos (Enterprise Web Beans)

The specification says that all EJB 3-style session and singleton beans are *enterprise* Web Beans. Message driven beans are not Web Beans # since they are not intended to be injected into other objects # but they can take advantage of most of the functionality of Web Beans, including dependency injection and interceptors.

Every local interface of an enterprise Web Bean that does not have a wildcard type parameter or type variable, and every one of its superinterfaces, is an API type of the enterprise Web Bean. If the EJB bean has a bean class local view, the bean class, and every one of its superclasses, is also an API type.

Stateful session beans should declare a remove method with no parameters or a remove method annotated `@Destructor`. The Web Bean manager calls this method to destroy the stateful session bean instance at the end of its lifecycle. This method is called the *destructor* method of the enterprise Web Bean.

```
@Stateful @SessionScoped
public class ShoppingCart {

   ...

   @Remove
   public void destroy() {}

}
```

Então, quando deveremos usar Web Bean corporativo (enterprise) em vez de um simples Web Bean? Bem, sempre que tivermos a necessidade de serviços corporatvios (enterprise) avançados oferecidos pelo EJB, tais como:

- gerenciamento de transações e segurança em nível de método,
- gerenciamento de concorrência,
- instance-level passivation for stateful session beans and instance-pooling for stateless session beans,
- remoto e invocação de web service, e
- temporizadores (timers) e métodos assíncronos

devemos utilizar um Web Bean corporativo (enterprise). Quando não precisamos de nenhuma destas coisas, um Web Bean simples vai servir muito bem.

Muitos Web Beans (incluindo qualquer Web Bean em escopo de sessão ou de aplicação) estão disponíveis para acesso concorrente. Por isso, o gerenciamento de concorrência fornecida pelo EJB 3.1 é especialmente útil. A maioria dos Web Beans em escopo de sessão e aplicação devem ser EJBs.

Web Beans which hold references to heavy-weight resources, or hold a lot of internal state benefit from the advanced container-managed lifecycle defined by the EJB `@Stateless`/`@Stateful`/`@Singleton` model, with its support for passivation and instance pooling.

Por último, isso normalmente é óbvio quando gerenciamento de transações e segurança em nível de método, temporizadores, métodos remotos ou assíncronos são necessários.

It's usually easy to start with simple Web Bean, and then turn it into an EJB, just by adding an annotation: `@Stateless`, `@Stateful` or `@Singleton`.

## 1.3.3. Métodos produtores (producer methods)

A *producer method* is a method that is called by the Web Bean manager to obtain an instance of the Web Bean when no instance exists in the current context. A producer method lets the application take full control of the instantiation process, instead of leaving instantiation to the Web Bean manager. For example:

```
@ApplicationScoped
public class Generator {

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

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

```
    }

}
```

O resultado do método produtor é injetado como qualquer outro Web Bean.

```
@Random int randomNumber
```

The method return type and all interfaces it extends/implements directly or indirectly are API types of the producer method. If the return type is a class, all superclasses are also API types.

Alguns métodos produtores retornam objetos que exigem destruição explícita :

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

Estes métodos produtores podem definir *métodos eliminação (disposal methods)*:

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

Este método de eliminação (disposal method) é chamado automaticamente pelo gerenciador do Web Bean no final da requisição.

Falaremos mais sobre métodos produtores no *Capítulo 6, Métodos produtores*.

### 1.3.4. JMS endpoints

Finally, a JMS queue or topic can be a Web Bean. Web Beans relieves the developer from the tedium of managing the lifecycles of all the various JMS objects required to send messages to queues and topics. We'll discuss JMS endpoints in *Seção 13.4, “Endpoints JMS”*.

# Exemplo de aplicação web JSF

Ilustraremos essas idéias com um exemplo completo. Nós implementaremos um login/logout de usuário de uma aplicação que utiliza JSF. Primeiro, definiremos um Web Bean que irá armazenar o nome do usuário (username) e a senha (password) fornecidos durante o login:

```
@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; }

}
```

Esse Web Bean é vinculado ao prompt de login do seguinte formulário 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
>
```

O verdadeiro trabalho é realizado por um Web Bean em escopo de sessão que mantém informações sobre o atual usuário conectado e expõe a entidade `User` para outros 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;
   }

}
```

Certamente, `@LoggedIn` é uma anotação de binding (binding annotation):

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

Agora, qualquer outro Web Bean pode facilmente injetar o usuário atual:

```
public class DocumentEditor {

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

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

}
```

Esperamos que esse exemplo tenha dado um gostinho do modelo de programação com Web Beans. No capítulo seguinte, exploraremos a injeção de dependência da Web Beans em profundidade.

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

A Web Beans está sendo desenvolvida no *projeto Seam* [http://seamframework.org/WebBeans]. Você pode baixar a última versão da Web Beans na *página de downloads* [http://seamframework.org/Download].

A Web Beans vem com duas aplicações de exemplo: `webbeans-numberguess` - um war, contendo apenas beans simples (simple beans) e `webbeans-translator`, e um ear, contendo beans corporativos (enterprise beans) -. Existem ,ainda, duas variações do exemplo numberguess: o exemplo tomcat (adequado para a implantação no Tomcat) e o exemplo jsf2, que você pode usar se estiver utilizando JSF2. Para executar os exemplos, você precisará do seguinte:

- a última versão da Web Beans,
- JBoss AS 5.0.1.GA, e
- Apache Tomcat 6.0.x, e
- Ant 1.7.0.

## 3.1. Utilizando o JBoss AS 5

Você precisará fazer o download do JBoss AS 5.0.1.GA em *jboss.org* [http://www.jboss.org/jbossas/downloads/] e descompactá-lo. Por exemplo:"

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

Depois, faça o download da Web Beans em *seamframework.org* [http://seamframework.org/Download] e descompacte-o. Por exemplo

```
$ cd ~/
$ unzip ~/webbeans-1.0.0.ALPHA1.zip
```

Em seguida, temos de dizer aos Web Beans onde o JBoss está localizado. Editar o `jboss-as/build.properties` e definir a propriedade `jboss.home`. Por exemplo:

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

Para instalar a Web Beans, você precisará do Ant 1.7.0 instalado e a variável de ambiente `ANT_HOME` setada. Por exemplo:

**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
```

Então, você pode instalar a atualização. O script de atualização utilizará o Maven para fazer o download da Web Beans automaticamente.

```
$ cd webbeans-1.0.0.ALPHA1/jboss-as
$ ant update
```

Agora, você está pronto para fazer a publicação do seu primeiro exemplo!

**Dica**

Os scripts para criar os exemplos oferecem uma série de alvos para JBoss AS, entre eles:

- `ant restart` - implanta o exemplo no formato explodido
- `ant explode` - atualiza o exemplo explodido, sem reiniciar
- `ant deploy` - implanta o exemplo no formato jar compactado
- `ant undeploy` - remove o exemplo do servidor
- `ant clean` - limpa o exemplo

Para implantar o exemplo numberguess:

```
$ cd examples/numberguess
```

```
ant deploy
```

Inicializando o JBoss AS:

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

**Dica**

Se você utiliza o Windows, utilize o script `run.bat`.

Aguarde até que a aplicação seja implantada, e desfrute de horas de diversão em *http://localhost:8080/webbeans-numberguess*!

Web Beans inclui um segundo exemplo simples que irá traduzir o seu texto para o Latim. O exemplo numberguess é um war e usa apenas beans simples; o exemplo translator é um exemplo ear e inclui benas corporativos, empacotados em um módulo EJB. Para experimentá-lo:

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

Aguarde até que a aplicação seja implantada, e acesse *http://localhost:8080/webbeans-translator*!

## 3.2. Utilizando o Apache Tomcat 6.0

Depois, faça o download do Tomcat 6.0.18 ou posterior em *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], e descompacte-o. Por exemplo

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

Depois, faça o download da Web Beans em *seamframework.org* [http://seamframework.org/Download] e descompacte-o. Por exemplo

```
$ cd ~/
$ unzip ~/webbeans-1.0.0.ALPHA1.zip
```

Em seguida, temos de dizer aos Web Beans onde o Tomcat está localizado. Editar o `jboss-as/build.properties` e definir a propriedade `tomcat.home`. Por exemplo:

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

**Dica**

Os scripts para criar os exemplos oferecem uma série de alvos para Tomcat. São eles:

- `ant tomcat.restart` - publica o exemplo no formato explodido
- `ant tomcat.explode` - atualiza o exemplo explodido, sem reiniciar
- `ant tomcat.deploy` - publica o exemplo no formato jar compactado
- `ant tomcat.undeploy` - remove o exemplo do servidor
- `ant tomcat.clean` - clean the example

Para implantar o exemplo numberguess no tomcat:

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

Inicializando o Tomcat:

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

**Dica**

Se você utiliza o Windows, utilize o script `startup.bat`.

Aguarde até que a aplicação seja implantada, e desfrute de horas de diversão em *http://localhost:8080/webbeans-numberguess*!

## 3.3. Utilizando o GlassFish

TODO

## 3.4. O exemplo numberguess

Na aplicação numberguess você terá 10 tentativas para adivinhar um número entre 1 e 100. Após cada tentativa, você será informado se você disse muito acima, ou muito abaixo.

O exemplo numberguess é composto de um número de Web Beans, arquivos de configuração e páginas Facelet JSF , empacotados como um war. Vamos começar com os arquivos de configuração.

Todos os arquivos de configuração para este exemplo estão localizados no `WEB-INF/`, que é armazenado no `WebContent` na árvore de fontes. Primeiro, temos `faces-config.xml`, onde dizemos para o JSF usar o 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
>
```

Existe um arquivo `web-beans.xml` vazio, que assinala essa aplicação como uma aplicação Web Beans.

Finalmente no `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.

No exemplo, existem 4 classes: as duas primeiras são tipos de binding. Primeiro, há o tipo de binding `@Random`, utilizado para a injeção de um número aleatório:

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

Há também o binding type `@MaxNumber`, utilizado para injetar o número máximo que pode ser injetado:

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

A classe `Generator` é responsável por criar um número aleatório, através de um método produtor. Ela também expõe o número máximo possível através de um método produtor:

```
@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;
  }

}
```

Você perceberá que o `Generator` está no escopo de aplicação; portanto, não obtemos um número aleatório diferente a cada vez.

O Web Bean final da aplicação é o `Game` em escopo de sessão.

Você notará que nós utilizamos a anotação `@Named`, para que possamos utilizar o bean através EL na página JSF. Finalmente, utilizamos injeção de construtor para inicializar o jogo com um número aleatório. E, claro, precisamos de dizer ao jogador quando ele venceu. Por isso, informaremos através do `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`.

**Dica**

Claro, você também deve incluir JSF e EL, bem como anotações comuns (common annotations) (`jsr250-api.jar`), que um servidor JEE inclui por padrão.

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.

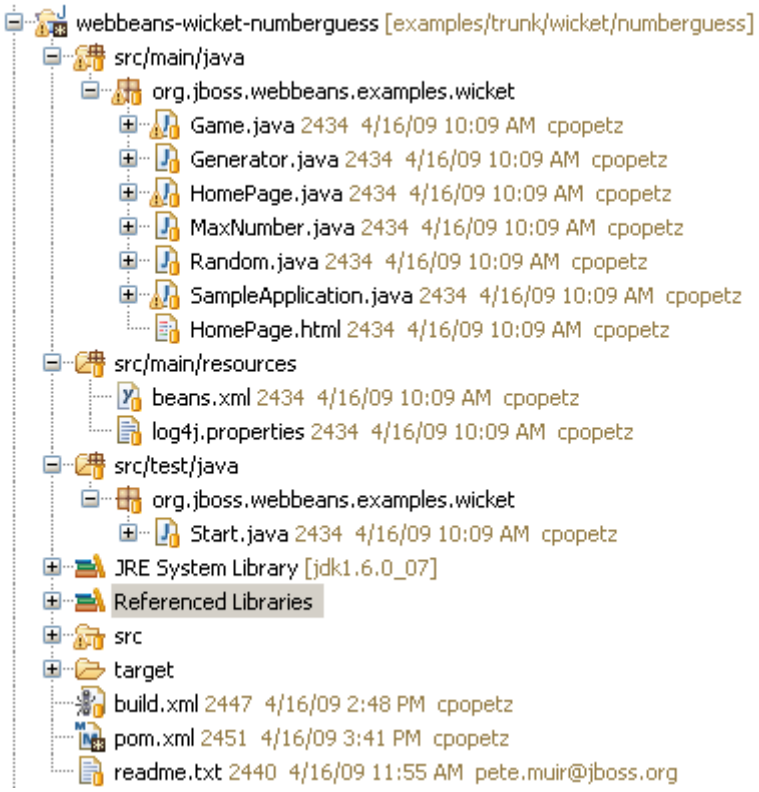
#### 3.4.2.1. Creating the Eclipse project

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. O exemplo translator

O exemplo translator pegará qualquer frase que você fornecer e a traduzirá para o Latim.

O exemplo translator é construído como um ear e contém EJBs. Como resultado, a sua estrutura é mais complexa do que o exemplo numberguess.

**Nota**

EJB 3.1 e Java EE 6 permitem a você empacotar EJBs em um war, o que tornará esta estrutura muito mais simples!

Primeiro, vamos dar uma olhada no ear aggregator, que está localizado módulo `webbeans-translator-ear`. Maven gera automaticamente `application.xml` para nós:

```
<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
>
```

Aqui nós definiremos o caminho do contexto, que nos dá uma url amigável (*http://localhost:8080/webbeans-translator*). ulink>) .

### Dica

Se você não está usando o Maven para gerar esses arquivos, você precisaria `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
>Ear Example for the reference implementation of 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
>
```

Em seguida, vamos ver o war. Tal como no exemplo numberguess, temos um `faces-config.xml` (para habilitar o Facelets) e um `web.xml` (para habilitar o JSF) no `WebContent/WEB-INF`.

Mais interessante é o facelet utilizado para traduzir texto. Tal como no exemplo numberguess, temos um template, que envolve o formulário (omitido aqui por brevitude):

```
<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
>
```

O usuário pode digitar um texto no textarea esquerdo e clicar no botão traduzir para ver o resultado à direita.

Por fim, vamos olhar o módulo EJB `webbeans-translator-ejb`. Em `src/main/resources/META-INF` existe apenas um `web-beans.xml` vazio, utilizado para marcar o arquivo como contendo Web Beans.

Deixamos o pedaço mais interessante para o final: o código! O projeto tem dois beans simples, `SentenceParser` e `TextTranslator` e dois beans corporativos`TranslatorControllerBean` e `SentenceTranslator`. Você deve estar bastante familiarizado com o que um Web Bean parece até agora. Então, vamos apenas destacar as partes mais interessantes aqui.

Tanto `SentenceParser` quanto `TextTranslator` são beans dependentes, e `TextTranslator` usa inicialização por construtor :

```
<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
>
```

`TextTranslator` é um bean stateless (com uma interface de negócios local), onde a mágica acontece . Obviamente, não poderíamos desenvolver um tradutor completo, mas lhe demos um bom caminho!

Finalmente, há um controlador orientado à interface, que recolhe o texto do usuário e despacha para o tradutor. Esse é um escopo de requisição, com o nome, stateful session bean, que injeta o tradutor.

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

  @Current TextTranslator translator;
```

O bean também tem getters e setters para todos os campos da página.

Como esse é um stateful session bean, temos de ter um método de remoção:

```
  @Remove
```

```
  public void remove()
  {

  }
```

O gerenciador do Web Beans chamará o método remover para você quando o bean for destruído - neste caso, no final da requisição.

Está encerrado o nosso curto passeio pelos exemplos de Web Beans. Para mais informações sobre a Web Beans, ou para ajudar, por favor visite *http://www.seamframework.org/WebBeans/Development*.

Precisamos de ajuda em todas as áreas - correção de bugs, escrita de novas funcionalidades, escrita de exemplos e tradução deste guia de referência.

# Injeção de Dependências

Web Beans suporta três principais mecanismos de injeção de dependências:

Injeção de parametros no construtor:

```
public class Checkout {

   private final ShoppingCart cart;

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

}
```

*Initializer* injeção por parâmetro de método:

```
public class Checkout {

   private ShoppingCart cart;

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

}
```

E injeção direta de campos:

```
public class Checkout {

   private @Current ShoppingCart cart;

}
```

A injeção de dependências sempre ocorre quando a instância do Web Bean é instanciada.

- Em primeiro lugar, a gerenciador do Web Bean chama o construtor do Web Bean para obter uma instância do Web Bean.
- Em seguida, o gerenciador do Web Bean inicializa os valores de todos os campos injetados do Web Bean.
- Em seguida, o gerenciador do Web Bean chama todos os métodos do Web Bean.
- Finalmente, o método `@PostConstruct` do Web Bean, se for o caso, é chamado.

Injeção de parâmetros no construtor não é suportado em EJB beans, uma vez que o EJB é instanciado pelo container EJB e não pelo gerenciador do Web Bean.

Parameters of constructors and initializer methods need not be explicitly annotated when the default binding type `@Current` applies. Injected fields, however, *must* specify a binding type, even when the default binding type applies. If the field does not specify a binding type, it will not be injected.

Métodos produtores também suportam injeção de parâmetros:

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

Por fim, métodos de observação (que iremos detalhar em *Capítulo 9, Eventos*), métodos de disposal e métodos destrutores, todos suportam injeção de parâmetros.

The Web Beans specification defines a procedure, called the *typesafe resolution algorithm*, that the Web Bean manager follows when identifying the Web Bean to inject to an injection point. This algorithm looks complex at first, but once you understand it, it's really quite intuitive. Typesafe resolution is performed at system initialization time, which means that the manager will inform the user immediately if a Web Bean's dependencies cannot be satisfied, by throwing a `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

O objetivo deste algoritmo é permitir que vários Web Beans implementem o mesmo tipo de API e também:

- permitir que o cliente escolha a implementação que lhe melhor convier utilizando *anotações de binding (binding annotations)*,
- permitir ao implantador (deployer) da aplicação escolher qual a implentação é adequada para uma determinada implantação, sem alterações para o cliente, por ativar ou desativar *tipos de implantação (deployment types)*, ou
- permitir uma implementação de uma API sobrescrever uma outra implementação da mesma API em tempo de implantação, sem alterações no cliente, utilizando *precedência do tipo de implantação (deployment type precedence)*.

Vamos explorer como o gerenciador do Web Beans determina qual o Web Bean deve ser injetado.

## 4.1. Anotações de ligação (binding annotations)

If we have more than one Web Bean that implements a particular API type, the injection point can specify exactly which Web Bean should be injected using a binding annotation. For example, there might be two implementations of `PaymentProcessor`:

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

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

Onde `@PayByCheque` e `@PayByCreditCard` são anotações de binding:

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

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

Um desenvolvedor cliente de um Web Bean utiliza a anotação de binding para especificar exatamente quais Web Bean devem ser injetados.

Utilizando injeção por campos (field injection):

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

Utilizando injeção de método de inicialização:

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

Ou utilizando injeção de construtor

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

### 4.1.1. Binding annotations with members

Binding annotations may have members:

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

Em cada caso, o valor do membro é significante:

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

You can tell the Web Bean manager to ignore a member of a binding annotation type by annotating the member `@NonBinding`.

### 4.1.2. Combinações de anotações de binding

Um ponto de injeção pode até mesmo especificar múltiplas anotações de binding:

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

Neste caso, só o Web Bean que tem *ambas* anotações de binding são elegíveis para injeção.

### 4.1.3. Anotações de binding e métodos produtores

Mesmo métodos produtores podem especificar anotações de binding:

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

### 4.1.4. O tipo padrão de binding

Web Beans defines a binding type `@Current` that is the default binding type for any injection point or Web Bean that does not explicitly specify a binding type.

Há duas situações comuns nas quais é necessário indicar explicitamente `@Current`:

- em um campo, a fim de declará-lo como um campo injetado com o tipo de binding padrão, e
- em um Web Bean, que tem um outro tipo de binding além do tipo padrão de binding.

## 4.2. Tipo de deploy

All Web Beans have a *deployment type*. Each deployment type identifies a set of Web Beans that should be conditionally installed in some deployments of the system.

For example, we could define a deployment type named `@Mock`, which would identify Web Beans that should only be installed when the system executes inside an integration testing environment:

```
@Retention(RUNTIME)
 @Target({TYPE, METHOD})
 @DeploymentType
 public @interface Mock {}
```

Suponha que temos alguns Web Beans que interajam com um sistema externo para processar pagamentos:

```
public class ExternalPaymentProcessor {

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

}
```

Uma vez que esse Web Bean não especifica explicitamente um tipo de implantação, ele tem o tipo de implantação padrão `@Production`.

For integration or unit testing, the external system is slow or unavailable. So we would create a mock object:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

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

}
```

But how does the Web Bean manager determine which implementation to use in a particular deployment?

## 4.2.1. Ativando tipos de implantação (deployment types)

Web Beans defines two built-in deployment types: `@Production` and `@Standard`. By default, only Web Beans with the built-in deployment types are enabled when the system is deployed. We can identify additional deployment types to be enabled in a particular deployment by listing them in `web-beans.xml`.

Going back to our example, when we deploy our integration tests, we want all our `@Mock` objects to be installed:

```
<WebBeans>
   <Deploy>
```

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

Now the Web Bean manager will identify and install all Web Beans annotated `@Production`, `@Standard` or `@Mock` at deployment time.

The deployment type `@Standard` is used only for certain special Web Beans defined by the Web Beans specification. We can't use it for our own Web Beans, and we can't disable it.

The deployment type `@Production` is the default deployment type for Web Beans which don't explicitly declare a deployment type, and may be disabled.

### 4.2.2. Precedencia dos tipos de deploy

If you've been paying attention, you're probably wondering how the Web Bean manager decides which implementation # `ExternalPaymentProcessor` or `MockPaymentProcessor` # to choose. Consider what happens when the manager encounters this injection point:

```
@Current PaymentProcessor paymentProcessor
```

There are now two Web Beans which satisfy the `PaymentProcessor` contract. Of course, we can't use a binding annotation to disambiguate, since binding annotations are hard-coded into the source at the injection point, and we want the manager to be able to decide at deployment time!

The solution to this problem is that each deployment type has a different *precedence*. The precedence of the deployment types is determined by the order in which they appear in `web-beans.xml`. In our example, `@Mock` appears later than `@Production` so it has a higher precedence.

Whenever the manager discovers that more than one Web Bean could satisfy the contract (API type plus binding annotations) specified by an injection point, it considers the relative precedence of the Web Beans. If one has a higher precedence than the others, it chooses the higher precedence Web Bean to inject. So, in our example, the Web Bean manager will inject `MockPaymentProcessor` when executing in our integration testing environment (which is exactly what we want).

It's interesting to compare this facility to today's popular manager architectures. Various "lightweight" containers also allow conditional deployment of classes that exist in the classpath, but the classes that are to be deployed must be explicity, individually, listed in configuration code or in some XML configuration file. Web Beans does support Web Bean definition and configuration via XML, but in the common case where no complex configuration is required, deployment types

allow a whole set of Web Beans to be enabled with a single line of XML. Meanwhile, a developer browsing the code can easily identify what deployment scenarios the Web Bean will be used in.

### 4.2.3. Exemplo de tipos de deploy

Deployment types are useful for all kinds of things, here's some examples:

- `@Mock` and `@Staging` deployment types for testing
- `@AustralianTaxLaw` for site-specific Web Beans
- `@SeamFramework`, `@Guice` for third-party frameworks which build on Web Beans
- `@Standard` for standard Web Beans defined by the Web Beans specification

I'm sure you can think of more applications...

## 4.3. Fixing unsatisfied dependencies

The typesafe resolution algorithm fails when, after considering the binding annotations and and deployment types of all Web Beans that implement the API type of an injection point, the Web Bean manager is unable to identify exactly one Web Bean to inject.

It's usually easy to fix an `UnsatisfiedDependencyException` or `AmbiguousDependencyException`.

To fix an `UnsatisfiedDependencyException`, simply provide a Web Bean which implements the API type and has the binding types of the injection point # or enable the deployment type of a Web Bean that already implements the API type and has the binding types.

To fix an `AmbiguousDependencyException`, introduce a binding type to distinguish between the two implementations of the API type, or change the deployment type of one of the implementations so that the Web Bean manager can use deployment type precedence to choose between them. An `AmbiguousDependencyException` can only occur if two Web Beans share a binding type and have exactly the same deployment type.

There's one more issue you need to be aware of when using dependency injection in Web Beans.

## 4.4. Proxies clientes

Clientes de um Web Bean injetado, geralmente não possuem uma referência direta a uma instância do Web Bean .

Imagine that a Web Bean bound to the application scope held a direct reference to a Web Bean bound to the request scope. The application scoped Web Bean is shared between many different requests. However, each request should see a different instance of the request scoped Web bean!

Now imagine that a Web Bean bound to the session scope held a direct reference to a Web Bean bound to the application scope. From time to time, the session context is serialized to disk in order to use memory more efficiently. However, the application scoped Web Bean instance should not be serialized along with the session scoped Web Bean!

Therefore, unless a Web Bean has the default scope `@Dependent`, the Web Bean manager must indirect all injected references to the Web Bean through a proxy object. This *client proxy* is responsible for ensuring that the Web Bean instance that receives a method invocation is the instance that is associated with the current context. The client proxy also allows Web Beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected Web Beans.

Unfortunately, due to limitations of the Java language, some Java types cannot be proxied by the Web Bean manager. Therefore, the Web Bean manager throws an `UnproxyableDependencyException` if the type of an injection point cannot be proxied.

Os seguintes tipos Java não podem ser "proxied" pelo gerenciador do Web Bean:

- classes que são declaradas `final` ou que tenham um método `final`,
- classes que não têm um construtor não privado sem parâmetros, e
- arrays e tipos primitivos.

It's usually very easy to fix an `UnproxyableDependencyException`. Simply add a constructor with no parameters to the injected class, introduce an interface, or change the scope of the injected Web Bean to `@Dependent`.

## 4.5. Obtendo um Web Bean via lookup programaticamente

A aplicação pode obter uma instância da interface `Manager` por injeção:

```
@Current Manager manager;
```

O objeto `Manager` fornece um conjunto de métodos para a obtenção de uma instância de um Web Bean programaticamente.

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

Binding annotations may be specified by subclassing the helper class `AnnotationLiteral`, since it is otherwise difficult to instantiate an annotation type in Java.

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

If the binding type has an annotation member, we can't use an anonymous subclass of `AnnotationLiteral` # instead we'll need to create a named subclass:

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

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

## 4.6. Chamadas ao ciclo de vida, `@Resource`, `@EJB` e `@PersistenceContext`

Enterprise Web Beans support all the lifecycle callbacks defined by the EJB specification: `@PostConstruct`, `@PreDestroy`, `@PrePassivate` and `@PostActivate`.

Web Beans simples suportam apenas chamadas a `@PostConstruct` e `@PreDestroy`.

Both enterprise and simple Web Beans support the use of `@Resource`, `@EJB` and `@PersistenceContext` for injection of Java EE resources, EJBs and JPA persistence contexts, respectively. Simple Web Beans do not support the use of `@PersistenceContext(type=EXTENDED)`.

A chamada ao `@PostConstruct` sempre ocorre após todas as dependências serem injetadas.

## 4.7. O objeto `InjectionPoint`

There are certain kinds of dependent objects # Web Beans with scope `@Dependent` # that need to know something about the object or injection point into which they are injected in order to be able to do what they do. For example:

- The log category for a `Logger` depends upon the class of the object that owns it.

- Injection of a HTTP parameter or header value depends upon what parameter or header name was specified at the injection point.
- Injeção do resultado da avaliação de uma expressão EL depende da expressão que foi especificada no ponto de injeção.

A Web Bean with scope `@Dependent` may inject an instance of `InjectionPoint` and access metadata relating to the injection point to which it belongs.

Let's look at an example. The following code is verbose, and vulnerable to refactoring problems:

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

This clever little producer method lets you inject a JDK `Logger` without explicitly specifying the log category:

```
class LogFactory {

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

}
```

Podemos agora escrever:

```
@Current Logger log;
```

Not convinced? Then here's a second example. To inject HTTP parameters, we need to define a binding type:

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

We would use this binding type at injection points as follows:

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

O seguinte método produtor faz o trabalho:

```
class HttpParams

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

}
```

(Note that the `value()` member of the `HttpParam` annotation is ignored by the Web Bean manager since it is annotated `@NonBinding`.)

The Web Bean manager provides a built-in Web Bean that implements the `InjectionPoint` interface:

```
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();
}
```

# Escopos e contextos

Até agora, vimos alguns exemplos de *anotações de tipo de escopo*. O escopo de um Web Bean determina o ciclo de vida das instâncias do Web Bean. O escopo também determina que clientes se referem a quais instâncias do Web Bean. De acordo com a especificação Web Beans, um escopo determina:

- Quando uma nova instância de qualquer Web Bean com esse escopo é criada
- Quando uma instância de qualquer Web Bean com esse escopo é destruída
- Cada referência injetada refere-se a qualquer instância de um Web Bean com esse escopo

Por exemplo, se temos um Web Bean de escopo de sessão `CurrentUser`, todos os Web Beans que são chamados no contexto dele `HttpSession` verão a mesma instância do `CurrentUser`. Essa instância será criada automaticamente na primeira vez que um `CurrentUser` for necessário nessa sessão, e será automaticamente destruída quando a sessão terminar.

## 5.1. Tipos de escopo

Web Beans possui um *modelo extensível de contexto*. É possível definir novos escopos, criando uma nova anotação de tipo de escopo:

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

Evidentemente, essa é a parte mais fácil do trabalho. Para esse tipo de escopo ser útil, nós também precisamos definir um objeto `Context` que implementa o escopo! Implementar um `Context` é geralmente uma tarefa muito técnica, destinada apenas ao desenvolvimento do framework.

Podemos aplicar uma anotação de tipo de escopo a uma classe de implementação de um Web Bean para especificar o escopo do Web Bean:

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

Normalmente, você usará um dos escopos pré-definidos na Web Bean.

## 5.2. Escopos pré-definidos

A Web Beans pré-define quatro tipos de escopos:

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

Para uma aplicação web que utiliza Web Beans:

- qualquer requisição servlet tem acesso aos escopos de requisição, sessão e aplicação ativos, e, adicionalmente
- qualquer requisição JSF tem acesso ao escopo de conversação ativo.

Os escopos de request e aplicação também estão disponíveis:

- durante invocações de métodos remotos de EJB,
- durante timeouts de EJB,
- durante a entrega de uma mensagem a um mensagem-driven bean, e
- durante a invocação de um web service

Se a aplicação tentar invocar um Web Bean com um escopo que não tem um contexto ativo, uma `ContextNotActiveException` é lançada pelo gerenciador do Web Bean em tempo de execução.

Três dos quatro escopos pré-definidos devem ser extremamente familiares a todos os desenvolvedores Java EE, então não vamos perder tempo discutindo-os aqui. Um dos escopos, porém, é novo.

## 5.3. O escopo de conversação

O escopo de conversação da Web Beans é um parecido com o tradicional escopo de sessão na medida em que mantém estado associado a um usuário do sistema, e o mantém durante várias requisições para o servidor. No entanto, ao contrário do escopo de sessão, o escopo de conversação:

- é demarcado explicitamente pela aplicação, e
- mantém o estado associado a uma determinada aba em um navegador web em uma aplicação JSF.

Uma conversação representa uma tarefa, uma unidade de trabalho do ponto-de-vista do usuário. O contexto de conversação mantém o estado associado ao usuário que estiver utilizando no momento. Se o usuário estiver fazendo várias coisas ao mesmo tempo, existem várias conversações.

A conversação está ativa durante qualquer requisição JSF. No entanto, a maioria das coversações é destruída no final da requisição. Se uma conversação deve manter estado através de múltiplas requisições, deve explicitamente ser promovida para uma *conversação de longa duração (long-running conversation)*.

### 5.3.1. Demarcação de contexto

Web Beans oferece um Web Bean pré-definido para o controle do ciclo de vida das conversações em uma aplicação JSF. Esse Web Bean pode ser obtido por injeção:

```
@Current Conversation conversation;
```

Para promover a conversação associada a requisição atual em uma conversação de longa duração, chame o método `begin()` no código da aplicação. Para agendar a destruição do atual contexto de conversão de longa duração no final da requisição atual, chame `end()`.

No exemplo a seguir, um Web Bean em escopo de conversação controla a conversação ao qual estiver associado:

```
@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() {}

}
```

Esse Web Bean é capaz de controlar seu próprio ciclo de vida através do uso da API `Conversation`. Mas alguns outros Web Beans têm um cliclo vida que depende totalmente de um outro objeto.

## 5.3.2. Propagação de conversação

Contexto de conversação propaga-se automaticamente em qualquer requisição faces JSF (formulário de submissão JSF). E não se propaga automaticamente em requisições não-faces, por exemplo, em navegação através de um link.

Nós podemos forçar a propagação da conversação em uma requisição não-faces incluindo o identificador único da conversação como um parâmetro da requisição. A especificação Web Beans reserva o parâmetro denominado `cid` para essa utilização. O identificador único da conversação pode ser obtido a partir do objeto `Conversation`, que tem o nome Web Bean `conversation`.

Portanto, o seguinte link propaga a conversação:

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

O gerenciador do Web Bean também é utilizado para propagar conversações em qualquer redirecionamento, mesmo se a conversação não estiver marcada como uma conversação de longa duração. Isso torna muito fácil a implementação do padrão POST-then-redirect, sem ter de recorrer a construções frágeis, como um objeto "flash". Neste caso, o gerenciador do Web Bean automaticamente adiciona um parâmetro de requisição a URL redirecionada.

## 5.3.3. Timeout de conversação

O gerenciador do Web Bean pode destruir uma conversação e todos os estados mantidos em seu contexto, a qualquer momento, a fim de preservar recursos. A implementação do gerenciador do Web Bean irá normalmente fazer isso, com base em algum tipo de timeout # embora isso não

seja exigido pela especificação Web Beans. O timeout é o período de inatividade antes que a conversação seja destruída.

O objeto `Conversation` fornece um método para definir o tempo limite (timeout). Essa é uma sugestão para o gerente do Web Bean, que é livre para ignorar essa configuração.

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

## 5.4. O dependent pseudo-scope

Além dos quatro escopos pré-definidos, Web Beans possui o chamado *dependent pseudo-scope*. Esse é o escopo padrão para um Web Bean que não declare explicitamente um tipo de escopo.

Por exemplo, esse Web Bean tem o tipo de escopo `@Dependent`:

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

Quando um ponto de injeção num Web Bean resolve para um Web Bean dependente, uma nova instância do Web Bean dependente é criada a cada vez que o primeiro Web Bean for instanciado. Instâncias de Web Beans dependentes nunca são compartilhadas entre diferentes Web Beans ou diferentes pontos de injeção. Eles são *objetos dependentes* de alguma outra instância de Web Bean.

Instâncias de Web Bean dependentes são destruídas quando a instância de que eles dependem é destruída.

Web Beans torna fácil a obtenção de uma instância dependente de uma classe Java ou um EJB, mesmo se a classe ou EJB já tiverem sido declarados como um Web Bean com outro tipo de escopo.

### 5.4.1. A anotação `@New`

A anotação de binding pré-definida `@New` permite a definição *implicita* de um Web Bean dependente em um ponto de injeção. Suponha que nós declaramos o seguinte campo injetado:

```
@New Calculator calculator;
```

Em seguida, um Web Bean com escopo `@Dependent`, tipo de binding `@New`, API do tipo `Calculator`, classe de implementação `Calculator` e tipo de implantação `@Standard` é definido implicitamente.

Isso é verdade mesmo se `Calculator` *já* estiver declarado com um tipo de escopo diferente, por exemplo:

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

Portanto, os seguintes atributos injetados obtêm uma instância diferente de `Calculator`:

```
public class PaymentCalc {

    @Current Calculator calculator;
    @New Calculator newCalculator;

}
```

O campo `calculator` tem uma instância de `Calculator` em escopo de conversação injetada. O campo `newCalculator` tem uma nova instância do `Calculator` injetada, com ciclo de vida que é vinculado à `PaymentCalc`.

Essa funcionalidade é particularmente útil em métodos produtores, como poderemos verificar no próximo capítulo.

# Métodos produtores

Métodos produtores permitem superarmos certas limitações que surgem quando o gerenciador do Web Bean, em vez da aplicação, é responsável por instanciar objetos. Eles são também a forma mais fácil de integrar os objetos que não são Web Beans ao ambiente Web Beans. (Veremos uma segunda abordagem em *Capítulo 12, Definindo Web Beans utilizando XML*.)

De acordo com a especificação:

> 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. Escopo de um método produtor

O escopo padrão dos métodos produtores é `@Dependent`, e, por isso ,serão chamados *toda vez* que o gerenciador do Web Bean injetar esse atributo ou qualquer outro atributo que resolve para o mesmo método produtor. Assim, pode haver várias instâncias do objeto `PaymentStrategy` para cada sessão do usuário.

Para mudar esse comportamento, nós podemos adicionar a anotação `@SessionScoped` ao método.

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

Agora, quando o método produtor é chamado, o `PaymentStrategy` retornado será associado ao contexto sessão. O método produtor não será invocado novamente na mesma sessão.

## 6.2. Injeção em métodos produtores

Existe um problema potencial com o código acima. As implementações de `CreditCardPaymentStrategy` são instanciadas utilizando o operador de Java `new`. Objetos instanciados diretamente pela aplicação não usufruem da injeção de dependência e não possuem interceptadores.

Se não é isso o que queremos, podemos utilizar a injeção de dependência no método produtor para obter instâncias do 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, o que se `CreditCardPaymentStrategy` é um Web Bean de escopo de requisição? Então o método produtor tem o efeito de "promover" a instância atual no escopo de requisição para o escopo de sessão. Isso certamente é um erro! O objeto no escopo de requisição será destruído pelo gerenciador do Web Bean antes de terminar a sessão, mas a referência ao objeto será deixada "presa" no escopo sessão. Esse erro *não* será detectado pelo gerenciador do Web Bean. Por isso, tome cuidado quando retornar instâncias de Web Bean em métodos produtores!

Existem pelo menos três maneiras de corrigirmos esse erro. Podemos alterar o escopo da implementação do `CreditCardPaymentStrategy`, mas isso poderia afetar outros clientes desse Web Bean. A mehor opção seria alterar o escopo do médoto produtor para `@Dependent` ou `@RequestScoped`.

Mas, uma solução mais comum é utilizar a anotação especial de binding `@New`

## 6.3. Uso do `@New` em métodos produtores

Considere o seguinte método produtor:

```
@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;
   }
```

```
}
```

Assim a nova instância *dependente* de `CreditCardPaymentStrategy` será criada, passada para o método produtor, retornada pelo método produtor e, finalmente, associada ao contexto de sessão. O objeto dependente não será destruído até que o objeto `Preferences` seja destruído, no término da sessão.

# Parte II. Desenvolvendo cÃ³digo fracamente acoplado

O primeiro grande tema da Web Beans é *fraco acoplamento*. Já vimos três meios de alcançar o fraco acoplamento:

- *tipos de implantação* habilitam o polimorfismo em tempo de implantação,
- *método produtores* habilitam o polimorfismo em tempo de execução, e
- *gerenciamento contextual do ciclo de vida* desacopla o ciclo de vida do Web Bean.

Essas técnicas servem para habilitar o fraco acoplamento entre o cliente e o servidor. O cliente não está mais fortemente acoplado a uma implementação de uma API, nem é obrigado a gerenciar o ciclo de vida do objeto servidor. Essa abordagem permite que *objetos stateful interajam como se fossem serviços*.

O fraco acoplamento torna o sistema mais *dinâmico*. O sistema pode responder a mudanças de uma maneira bem definida. No passado, frameworks que tentaram prover essas facilidades acima listadas, invariavelmente acabaram sacrificando a type safety. A Web Beans é a primeira tecnologia que alcança esse nível de fraco acoplamento de uma meneira typesafe.

Web Beans provê três facilidades extras importantes que ultrapassam o objetivo do fraco acoplamento:

- *interceptadores* desacomplam detalhes técnicos da lógica de negócios,
- *decoradores* podem ser utilizados para desacoplar detalhes de negócios, e
- *notificadores de eventos* desacoplam os produtores de eventos dos consumidores de eventos.

Primeiramente, exploraremos os interceptadores

# Interceptadores

Web Beans re utiliza a arquitetura básica do interceptor de EJB 3.0, que extende a funcionalidade em duas direções:

- Qualquer Web Bean pode ter interceptores, não apenas session beans.
- Web Beans possui uma abordagem mais sofisticadas baseada em anotações para associar interceptores aos Web Beans.

A especificação de EJB define dois tipos de pontos de interceptação:

- interceptação de métodos de negócios, e
- interceptadores de chamadas de ciclo de vida

A *business method interceptor* applies to invocations of methods of the Web Bean by clients of the Web Bean:

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

A *lifecycle callback interceptor* applies to invocations of lifecycle callbacks by the container:

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

An interceptor class may intercept both lifecycle callbacks and business methods.

## 7.1. Bindings de interceptadores

Suppose we want to declare that some of our Web Beans are transactional. The first thing we need is an *interceptor binding annotation* to specify exactly which Web Beans we're interested in:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Now we can easily specify that our `ShoppingCart` is a transactional object:

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

Or, if we prefer, we can specify that just one method is transactional:

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

## 7.2. Implementando interceptadores (interceptors)

That's great, but somewhere along the line we're going to have to actually implement the interceptor that provides this transaction management aspect. All we need to do is create a standard EJB interceptor, and annotate it `@Interceptor` and `@Transactional`.

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

All Web Beans interceptors are simple Web Beans, and can take advantage of dependency injection and contextual lifecycle management.

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

   @Resource Transaction transaction;

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

}
```

Multiple interceptors may use the same interceptor binding type.

## 7.3. Habiliatando interceptadores (interceptors)

Finalmente, temos que *ativar* nossos interceptadores no `web-beans.xml`.

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

Whoah! Why the angle bracket stew?

Bem, a declaração XML resolve dois problemas:

- o que nos permite especificar totalmente a ordem para todos os interceptores em nosso sistema, garantindo um comportamento determinístico, e
- it lets us enable or disable interceptor classes at deployment time.

For example, we could specify that our security interceptor runs before our `TransactionInterceptor`.

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

Or we could turn them both off in our test environment!

## 7.4. Interceptor bindings with members

Suponhamos que queremos acrescentar algumas informações adicionais para o nossa anotação `@Transactional`:

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

Web Beans will use the value of `requiresNew` to choose between two different interceptors, `TransactionInterceptor` and `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew=true) @Interceptor
```

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

Now we can use `RequiresNewTransactionInterceptor` like this:

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

But what if we only have one interceptor and we want the manager to ignore the value of `requiresNew` when binding interceptors? We can use the `@NonBinding` annotation:

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

## 7.5. Multiple interceptor binding annotations

Usually we use combinations of interceptor bindings types to bind multiple interceptors to a Web Bean. For example, the following declaration would be used to bind `TransactionInterceptor` and `SecurityInterceptor` to the same Web Bean:

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

However, in very complex cases, an interceptor itself may specify some combination of interceptor binding types:

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

Then this interceptor could be bound to the `checkout()` method using any one of the following combinations:

```
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. Interceptor binding type inheritance

One limitation of the Java language support for annotations is the lack of annotation inheritance. Really, annotations should have reuse built in, to allow this kind of thing to work:

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

Well, fortunately, Web Beans works around this missing feature of Java. We may annotate one interceptor binding type with other interceptor binding types. The interceptor bindings are transitive # any Web Bean with the first interceptor binding inherits the interceptor bindings declared as meta-annotations.

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

Any Web Bean annotated `@Action` will be bound to both `TransactionInterceptor` and `SecurityInterceptor`. (And even `TransactionalSecureInterceptor`, if it exists.)

## 7.7. Use of `@Interceptors`

The `@Interceptors` annotation defined by the EJB specification is supported for both enterprise and simple Web Beans, for example:

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

However, this approach suffers the following drawbacks:

- the interceptor implementation is hardcoded in business code,
- interceptors may not be easily disabled at deployment time, and
- the interceptor ordering is non-global # it is determined by the order in which interceptors are listed at the class level.

Therefore, we recommend the use of Web Beans-style interceptor bindings.

# Decoradores

Interceptadores são um meio poderoso para capturar e separar preocupações *ortogonais* para o tipo de sistema. Qualquer interceptador é capaz de interceptar invocações de qualquer tipo Java. Isso os torna ideais para resolver questões técnicas, tais como gerenciamento de transação e segurança. No entanto, por natureza, interceptadores desconhecem a real semântica dos eventos que interceptam. Assim, interceptadores não são um instrumento adequado para a separação de questões relacionadas a negócios.

O contrário é verdadeiro *decoradores*. O decorador intercepta invocações apenas para uma determinada interface Java e, portanto, é ciente de toda a semântica que acompanha a interface. Isso torna os decoradores uma ferramenta perfeita para modelar alguns tipos de questões de negócios. Significa também que um decorador não tem a generalidade de um interceptador. Decoradores não são capazes de resolver questões técnicas que atravessam muitos tipos diferentes.

Suponha que temos uma interface que represente contas:

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

Vários Web Beans em nosso sistema implementam a interface `Account`. No entanto, temos uma obrigação legal que, para qualquer tipo de conta, as grandes transações devem ser registadas pelo sistema, em um registro especial (log). Esse é um trabalho perfeito para um decorador.

Um decorador é um Web Bean simples que implementa o tipo que decora e é anotado com `@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) );
        }
    }

}
```

Ao contrário de outros Web Beans simples, um decorador pode ser uma classe abstrata. Se não há nada de especial que o decorador precisa fazer para um determinado método da interface decorada, você não precisa implementar esse método.

## 8.1. Atributos delegados

Todos os decoradores têm um *atributo delegado*. O tipo e os tipos de binding do atributo delegado determinam a qual Web Beans o decorador está vinculado. O tipo do atributo delegado deve implementar ou estender todas as interfaces implementadas pelo decorador.

Este atributo delegado especifica que o decorador está vinculado a todos os Web Beans que implementam `Account`:

```
@Decorates Account account;
```

Um atributo delegado pode especificar uma anotação de binding. Então, o decorador só será vinculado ao Web Beans com o mesmo vínculo.

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

Um decorador é vinculado a qualquer Web Bean que:

- tenha como tipo do atributo delegado uma API, e
- tenha todos os tipos de vínculo que são declarados pelo atributo delegado.

O decorador pode invocar o atributo delegado,o que praticamente equivale a chamar `InvocationContext.proceed()` a partir de um interceptador

## 8.2. Habilitando decoradores

Nós precisamos *habilitar* nosso decorador no `web-beans.xml`.

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

Essa declaração tem o mesmo propósito para decoradores que a `<Interceptors>` tem para os interceptadores:

- isso possibilita-nos determinar a ordem total para todos os decoradores em nosso sistema, assegurando um comportamento determinístico
- isso permite habilitarmos ou desabilitarmos as classes decoradas em tempo de implantação.

Interceptadores para o método são chamados antes dos decoradores que se aplicam a esse método.

# Eventos

A habilidade de notificação de eventos da Web Beans permite aos Web Beans interagirem de maneira totalmente desacoplada. *Produtores* de eventos disparam eventos que são entregues aos *observadores* de eventos pelo gerenciador do Web Bean. Esse esquema básico pode soar como o familiar padrão observador/observável padrão, mas há várias diferenças:

- não só os produtores são desacoplados dos observadores; os observadores são totalmente desacoplados dos produtores,
- os observadores podem especificar uma combinação de "seletores" para reduzir o conjunto de eventos que irão receber notificações, e
- os observadores podem ser notificados imediatamente ou podem especificar que a notificação do evento deveria esperar até o término da transação corrente

## 9.1. Observadores de eventos

Um *método observador (observer method)* é um método de um Web Bean com um parâmetro anotado`@Observes`.

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

O parâmetro anotado é chamado *parâmetro de evento*. O tipo do parâmetro de evento é observado pelo *event type*. Os métodos observadores também podem especificar "seletores", que são apenas instâncias de tipo de bindings de Web Beans. Quando um tipo de binding é utilizado como um seletor de evento, é chamado de *tipo de binding de evento*.

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

Especificamos os bindings de eventos do observador, anotando o parâmetro do evento:

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

Um método observador não necessita especificar nenhum binding de evento # nesse caso, ele está interessado em *todos* os eventos de um determinado tipo. Se ele especificar o bindings de evento, ele estará apenas interessado em eventos que também têm esses bindings de evento.

O método observador pode ter parâmetros adicionais que são injetados de acordo com a semântica usual de injeção de parâmetros em métodos de Web Beans

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

## 9.2. Produtores de Eventos

O evento produtor pode obter um objeto *notificador de evento (event notifier)* por injeção:

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

A anotação `@Observable` define, implicitamente, uma Web Bean com o escopo `@Dependent` e tipo de implantação `@Standard`, com uma implementação fornecida pelo gerenciador do Web Bean.

Um produtor lança eventos chamando o método `fire()` da interface `Event` e passando um objeto *event object*:

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

Um objeto de evento pode ser uma instância de qualquer classe Java que não tem nenhum tipo de variáveis ou parâmetros tipo curinga. O evento será entregue a cada método observador que:

- tenha um parâmetro evento em que o evento objeto é atribuído, e
- não especifique nenhum evento bindings.

O gerenciador do Web Bean simplesmente chama todos os métodos de observação, passando o objeto do evento como o valor do parâmetro do evento. Se algum método observador lança uma exceção, o gerenciador do Web Bean deixa de chamar os métodos do observador, e a exceção é relançada pelo método `fire()`.

Para especificar um "seletor", o produtor do evento pode passar uma instância do tipo de binding do evento para o método `fire()`:

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

O classe auxiliar `AnnotationLiteral` permite instanciar tipo de binding inline, o que é difícil de se fazer em Java.

O evento será entregue a todo método observador (observer method) que:

- tenha um parâmetro evento em que o evento objeto é atribuído, e
- não especifique nenhum binding de evento *exceto* para o binding de evento passado para o `fire()`.

Alternativamente, bindings de eventos podem ser especificados anotando o ponto de injeção do notificador do evento:

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

Em seguida, todos os eventos disparados por essa instância de `Event` tem o binding de evento anotada. O evento será entregue a cada método observador que:

- tenha um parâmetro evento em que o evento objeto é atribuído, e
- não especifique nenhum binding de evento *exceto* para os bindings de evento passados para o `fire()` ou os bindings de evento anotados do ponto de injeção do notificador de evento.

## 9.3. Resgistrando observadores (observers) dinamicamente

Frequentemente, é útil registrar um evento observador dinamicamente. A aplicação pode implementar a interface `Observer` e registrar a instância com um evento notificador chamando o método `observe()`.

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

Tipos de binding de eventos podem ser especificados pelo notificador do evento no ponto de injeção, ou passando instâncias do tipo de binding de evento para o método observador `observe()` method:

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

## 9.4. Bindings de eventos com os membros

Um tipo de binding de evento pode ter anotações membro:

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Role {
   RoleType value();
}
```

O valor do membro é utilizado para reduzir as mensagens entregues ao observador:

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

Membros de tipo de binding de evento podem ser especificados estaticamente pelo produtor do evento, por meio de anotações no ponto de notificação do evento:

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

Alternativamente, o valor do membro do tipo de binding de evento pode ser determinado dinamicamente pelo produtor do evento. Vamos começar escrevendo uma subclasse abstrata de `AnnotationLiteral`:

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

O produtor do evento (event producer) passa uma instância dessa classe para `fire()`:

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

## 9.5. Múltiplos bindings de eventos

Tipos de binding de evento podem ser combinados, por exemplo:

```
@Observable @Blog Event<Document
> blogEvent;
```

```
...
if (document.isBlog()) blogEvent.fire(document, new AnnotationLiteral<Updated
>(){});
```

Quando esse evento ocorre, todos os métodos observadores que seguem esse evento serão 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 transacionais

Observadores transacionais recebem notificações de eventos durante, antes ou após a conclusão da transação em que o evento foi disparado. Por exemplo: o seguinte método observador necessita atualizar um conjunto de resultados de uma consulta que está armazenada no contexto da aplicação, mas apenas quando as transações que atualizam a àrvore de `Category`forem concluídas com sucesso:

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

Existem três tipos de observadores transacionais:

- `@AfterTransactionSuccess` observadores são chamados durante a fase após a conclusão da transação, mas somente se a transação tiver sido concluída com sucesso
- `@AfterTransactionFailure` observadores são chamados durante a fase após a conclusão da transação, mas somente se a transação não tiver sido concluída com sucesso
- `@AfterTransactionCompletion` observadores são chamados durante a fase após a conclusão da transação

- `@BeforeTransactionCompletion` observadores são chamados durante a fase antes da conclusão da transação

Observadores transacionais são muito importantes para um modelo de objetos stateful como o Web Beans, porque o estado é muitas vezes mantido por mais de uma única transação atômica.

Imagine que fizemos cache do conjunto de resultados da consulta JPA no escopo de aplicação (application scope):

```
@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 tempos em tempos, um `Product` é criado ou excluído. Quando isso ocorre, é preciso atualizar o catálogo de `Product`. Mas devemos esperar até *depois* da transação ser concluída com sucesso antes de realizar essa atualização!

O Web Bean que cria e remove `Products` pode lançar eventos, por exemplo:

```
@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
>(){});
    }

    ...

}
```

E agora `Catalog` pode observar os eventos após o término da transação concluída com sucesso:

```
@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. Obtendo o mÃ¡ximo da tipificaÃ§Ã£o forte

O segundo grande tema da Web Beans é a *tipificação forte*. As informações sobre as dependências, interceptores e decoradores de um Web Bean, e as informações sobre os consumidores de eventos para um produtor de evento, estão contidas em construtores Java typesafe, que podem ser validados pelo compilador.

Você não vê identificadores baseados em strings no código Web Beans - não porque o framework está escondendo-os de você utilizando padrões de regras inteligentes # o chamado "configuração por convenção" -, mas porque simplesmente não existem strings ali!

A vantagem óbvia dessa abordagem é que *qualquer* IDE pode fornecer auto completion, validação e refactoring sem necessidade de ferramentas especiais. Mas há uma segunda vantagem, menos imediatamente óbvia. Acontece que quando você começar a pensar na identificação de objetos, eventos ou interceptadores por meio de anotações - em vez de nomes -, você tem uma oportunidade para aumentar o nível semântico do seu código.

Web Beans incentiva você a desenvolver anotações que modelam conceitos. Por exemplo:

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

em vez de utilizar nomes compostos, como

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

As anotações são reutilizáveis. Elas ajudam a descrever qualidades comuns de partes diferentes do sistema. Elas nos ajudam a categorizar e entender o nosso código. Elas nos ajudam a lidar com questões comuns, de uma maneira comum. Elas tornam o nosso código mais legível e mais compreensível.

*Estereótipos* Web Beans levam essa idéia um pouco mais longe. Um estereótipo modela um *papel* comum na sua arquitetura de aplicação. Ele incorpora várias propriedades do papel - incluindo o escopo, bindings de interceptadores, tipos de implantação, etc - em um único pacote reutilizável.

Mesmo os metadados da Web Beans são fortemente tipados! Não há um compilador para XML, então a Web Beans tira proveito de esquemas XML para validar os tipos Java e os atributos que aparecem em XML. Essa abordagem acaba por tornar o XML mais legível, assim como anotações deixam nosso código Java mais legível.

Nós agora estamos prontos para verificar mais algumas funcionalidades avançadas da Web Beans. Tenha em mente que essas funcionalidades existem para tornar nosso código fácil para validar e, ao mesmo tempo, mais fácil de entender. Na maioria das vezes você nem *precisa* se preocupar em utilizar essas funcionalidades, mas se forem fáceis de usar, você apreciará seu poder.

# Estereótipos

De acordo com a especificação Web Beans:

> Em muitos sistemas, a utilização de padrões arquiteturais produz um conjunto de papéis Web Bean recorrentes. Um estereótipo permite a um desenvolvedor de framwework identificar esse papel e declarar alguns metadados comums para Web Beans com esse papel em um local centralizado.
>
> Um estereótipo encapsula qualquer combinação de:
>
> - um tipo padrão de implantação,
> - um tipo de escopo padrão,
> - uma restrição ao escopo do Web Bean,
> - uma exigência de que o Web Bean implemente ou estenda um certo tipo, e
> - um conjunto de anotações para binding de interceptadores
>
> Um estereótipo também pode especificar que todos os Web Beans com o estereótipo têm um nome Web Bean padrão.
>
> Um Web Bean pode declarar zero, um ou vários estereótipos.

Um estereótipo é um tipo de anotação Java. Esse estereótipo identifica classes de ação em algum framework MVC:

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

Nós utilizamos o estereótipo, aplicando a anotação ao Web Bean.

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

## 10.1. Escopo padrão e o tipo de implantação para um estereótipo

Um estereótipo pode especificar um escopo padrão e/ou um tipo padrão de implantação para Web Beans com esse estereótipo. Por exemplo, o tipo de implantação `@WebTier` identifica Web

Beans que só deverão ser implantados quando o sistema executar como uma aplicação web. Podemos especificar os seguintes padrões para classes de ação :

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

Evidentemente, uma determinada ação pode ainda,se necessário, substituir estes padrões:

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

Se quisermos forçar todas as ações para um escopo particular, podemos fazer isso também.

## 10.2. Restringindo o escopo e o tipo com um estereótipo

Suponha que queremos impedir as ações de declarar certos escopos. Web Beans permite-nos indicar explicitamente o conjunto de escopos permitidos para Web Beans com um certo estereótipo. Por exemplo:

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

Se uma determinada classe de ação tenta especificar um escopo diferente do escopo de requisição da Web Beans, uma exceção será lançada pelo gerenciador do Web Bean em tempo de inicialização.

Também podemos forçar todos os Web Beans com um certo estereótipo a implementar uma interface ou estender uma classe:

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

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

Se uma determinada classe de ação não estender a classe `AbstractAction`, uma exceção será lançada pelo gerenciador do Web Bean em tempo de inicialização.

## 10.3. Bindings de interceptadores para estereótipos

Um estereótipo pode especificar um conjunto de interceptadores de bindings a serem herdados por todos os Web Beans com esse estereótipo.

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

Isso nos ajuda a manter os detalhes técnicos ainda mais longe do código de negócios!

## 10.4. Padronização de nomes com estereótipos

Por último, é possível especificar que todos os Web Beans com um certo estereótipo tenham um nome Web Bean padronizado pelo gerenciador do Web Bean. As ações são,muitas vezes, referenciadas em páginas JSP. Por isso, elas são um caso de utilização perfeito desse recurso. Tudo o que precisamos fazer é adicionar uma anotação `@Nome` vazia:

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

Agora, `LoginAction` terá o nome `loginAction`.

## 10.5. Estereótipos padrões

Já conhecemos dois estereótipos padrões definidos pela especificação de Web Beans: `@Interceptor` e `@Decorator`.

Web Beans define mais um estereótipo padrão:

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

Esse estereótipo é destinado ao uso com o JSF. Em vez de utilizar JSF managed beans, basta anotar um Web Bean com `@Model`, e utilizá-lo diretamente em sua página JSF.

# Especialização

Nós já vimos a forma como o modelo de injeção de dependências da Web Beans permite *sobrescrever* a implementação da API em tempo de implantação. Por exemplo, o seguinte Bean Web corporativo fornece uma implementação da API `PaymentProcessor` em produção:

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

But in our staging environment, we override that implementation of `PaymentProcessor` with a different Web Bean:

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

What we've tried to do with `StagingCreditCardPaymentProcessor` is to completely replace `AsyncPaymentProcessor` in a particular deployment of the system. In that deployment, the deployment type `@Staging` would have a higher priority than the default deployment type `@Production`, and therefore clients with the following injection point:

```
@CreditCard PaymentProcessor ccpp
```

Pode receber uma instância de `StagingCreditCardPaymentProcessor`.

Infelizmente, existem várias armadilhas que facilmente podemos cair:

- the higher-priority Web Bean may not implement all the API types of the Web Bean that it attempts to override,
- the higher-priority Web Bean may not declare all the binding types of the Web Bean that it attempts to override,
- the higher-priority Web Bean might not have the same name as the Web Bean that it attempts to override, or

- the Web Bean that it attempts to override might declare a producer method, disposal method or observer method.

Em cada um destes casos, o Web Bean que tentamos sobrescrever ainda podia ser chamado em tempo de execução. Portanto, a sobrescrita é algo propensa a erros de desenvolvimento.

Web Beans provides a special feature, called *specialization*, that helps the developer avoid these traps. Specialization looks a little esoteric at first, but it's easy to use in practice, and you'll really appreciate the extra security it provides.

## 11.1. Utilizando a especialização

Specialization is a feature that is specific to simple and enterprise Web Beans. To make use of specialization, the higher-priority Web Bean must:

- ser uma subclasse direta do Web Bean que sobrescreve, e
- be a simple Web Bean if the Web Bean it overrides is a simple Web Bean or an enterprise Web Bean if the Web Bean it overrides is an enterprise Web Bean, and
- será anotada `@Specializes`.

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

Nós dizemos que a alta prioridade na Web Bean *specializa* sua superclasse.

## 11.2. Vantagens da especialização

Quando a especialização é utilizada:

- the binding types of the superclass are automatically inherited by the Web Bean annotated `@Specializes`, and
- the Web Bean name of the superclass is automatically inherited by the Web Bean annotated `@Specializes`, and
- métodos produtores, métodos de eliminação e métodos observadores declarados pela superclasse são chamados sobre uma instância do Web Bean anotado com `@Specializes`.

Em nosso exemplo, o tipo de ligação (binding type) `@CreditCard` do `CreditCardPaymentProcessor` é herdado por `StagingCreditCardPaymentProcessor`.

Além disso, o gerenciador do Web Bean irá validar que:

- all API types of the superclass are API types of the Web Bean annotated `@Specializes` (all local interfaces of the superclass enterprise bean are also local interfaces of the subclass),
- o tipo de implantação do Web Bean anotado com `@Specializes` tem uma precedência maior do que o tipo de implantação da superclasse, e
- não há outro Web Bean ativado que também especializa a superclasse.

Se qualquer uma dessas condições são violadas, o gerenciador do Web Bean lança uma excepção em tempo de inicialização.

Therefore, we can be certain that the superclass will *never* be called in any deployment of the system where the Web Bean annotated `@Specializes` is deployed and enabled.

# Definindo Web Beans utilizando XML

Até agora,vimos muitos exemplos de declaração de Web Beans usando anotações. No entanto, há várias situações em que não podemos usar anotações para definir um Web Bean:

- quando a classe de implementação vem de alguma biblioteca preexistente, ou
- quando deveria haver vários Web Beans com a mesma classe de implementação.

Em ambos os casos, We Beans nos dá duas opções:

- escrever um método produtor (producer method), ou
- declarar um Web Bean utilizando XML.

Muitos frameworks usam XML para fornecer metadados relativos às classes Java. No entanto, Web Beans utiliza uma abordagem muito diferente para especificar os nomes de classes Java, atributos ou métodos dos outros frameworks. Em vez de escrever os nomes das classes e dos membros ( como uma String de valores em elementos e atributos XML), Web Beans permite que você use o nome da classe ou membro como o nome do elemento XML.

A vantagem dessa abordagem é que você pode escrever um esquema XML (XML schema) que evita erros ortográficos no seu documento XML. É até mesmo possível para uma ferramenta gerar o esquema XML (XML schema) automaticamente, a partir do código Java compilad.Ou, um ambiente integrado de desenvolvimento poderia fazer a mesma validação sem a necessidade explicíta do passo intermediário de geração.

## 12.1. Declarando classes Web Beans

Para cada pacote Java, Web Beans define um namespace XML correspondente. O nome é formado precedendo `urn:java:` ao nome do pacote Java. Para o pacote `com.mydomain.myapp`, o namespace XML é `urn:java:com.mydomain.myapp`.

Tipos Java pertencentes a um pacote são referenciados a utilizar um elemento XML no namespace correspondente ao pacote. O nome do elemento é o nome do tipo Java. Atributos e métodos do tipo são especificados por elementos filhos do mesmo namespace. Se o tipo for uma anotação, os membros são definidos por atributos do elemento.

Por exemplo, o elemento `<util:Date/>` no seguinte fragmento XML refere-se à classe `java.util.Date`:

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

   <util:Date/>
```

```
</WebBeans
>
```

E esse é todo o código necessário para declarar que `Date` é um simples Web Bean! Uma instância de `Date` pode agora ser injetada por qualquer outro Web Bean:

```
@Current Date date
```

## 12.2. Declarando metadados Web Bean

Podemos declarar o escopo, tipo de puplicação (deployment type) e tipo de ligação de interceptador (interceptor binding types) com a utilização direta de elementos filhos na declaração do Web Bean:

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

Utilizamos exatamente a mesma abordagem para especificar nomes e tipos de ligação:

```
<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
>
```

Em que `@Login` e `@SystemStart` são anotações do tipo ligação.

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

Como é habitual, um Web Bean pode suportar múltiplos tipos de ligação (binding types):

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

Interceptadores e Decoradores são simplesmente Web Beans. Assim, podem ser declarados como qualquer outro Web Bean:

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

## 12.3. Declarando membros Web Bean

TODO!

## 12.4. Declarando inline Web Beans

Web Beans nos permite definir um Web Bean em um ponto de injeção. Por exemplo:

```
<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
>
```

O elemento `<Name>` declara um Web Bean simples de escopo `@Dependent` e classe `Name`, com um conjunto inicial de valores para os campos. Esse Web Bean possui a especial, container-generated binding and is therefore injectable only to the specific injection point at which it is declared.

Esse simples - mas poderoso - recurso permite que o formato XML do Web Beans seja utilizado para especificar grafos completos de objetos Java. Não é uma solução de ligação de dados (databinding) completa, mas está bem próxima!

## 12.5. Utilizando um esquema

Se desejamos que o formato do documento XML seja criado por pessoas que não são desenvolvedores Java, ou por pessoas que não têm acesso ao nosso código, precisamos fornecer um esquema (XML schema). Não há nada específico no Web Beans sobre escrever ou utilizar o 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
>
```

Escrever um esquema XML (XML schema) é um tanto tedioso. Tod via, o projeto Web Beans RI fornecerá uma ferramenta que automaticamente gera o esquema XML (XML schema), a partir do código da classe Java compilada.

# Parte IV. Web Beans e o ecossistema Java EE

A terceira motivação da Web Beans é *integração*. Web Beans foi projetada para trabalhar em conjunto com outras tecnologias, ajudando o desenvolvedor a trabalhar outras tecnologias conjuntamente. Web Beans é uma tecnologia aberta. Ela faz parte do ecossistema Java EE, e é por si só a base para um novo ecossistema de extensões portáveis e integração com os frameworks e as tecnologias existentes.

Nós já vimos como Web Beans ajuda a integrar EJB e JSF, permitindo que EJBs sejam associados diretamente a páginas JSF . Isso é só o começo. Web Beans oferece o mesmo potencial para diversas outras tecnologias, tais como motores de Gerenciamento de Processos de Negócios, outros Frameworks Web e modelos de componentes de terceiros. A plataforma Java EE nunca será capaz de padronizar todas as tecnologias interessantes que são utilizadas no mundo de desenvolvimento de aplicações Java, mas a Web Beans facilita a utilização das tecnologias que ainda não fazem parte da plataforma suavemente dentro do ambiente Java EE.

Estamos prestes a ver como tirar o máximo proveito da plataforma Java EE em uma aplicação que utiliza Web Beans. Reuniremos, brevemente, um conjunto de SPIs que são fornecidas para suportar extensões portáveis para Web Beans. Talvez você nunca precisará usar essas SPIs diretamente, mas é bom saber que estão lá se você precisar delas. Mais importante: você tirará proveito delas indiretamente, toda vez que você utilizar uma extensão de terceiros.

# Integração com o Java EE

A Web Beans está plenamente integrada ao ambiente Java EE. A Web Beans tem acesso aos recursos Java EE e aos contextos de persistência JPA. Eles podem ser utilizados em expressões EL Unificadas (Unified EL) e em páginas JSF e JSP. Podem até ser injetados em objetos que não são Web Benas, tais como Servlets e Message-Driven Beans.

## 13.1. Injetando recursos Java EE em um Web Bean

Todos Web Beans, simples e corporativos (enterprise Web Beans), podem usufruir da injeção de dependência do Java EE utilizando `@Resource`, `@EJB` e `@PersistenceContext`. Nós já vimos vários exemplos disso, embora não demos muita ênfase até o 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;

   ...

}
```

As chamadas Java EE `@PostConstruct` e `@PreDestroy` também são suportadas para todos os Web Beans simples e corporativos. O método anotado com `@PostConstruct` é invocado após *todas* injeções serem realizadas.

Existe uma restrição de que devemos estar conscientes: `@PersistenceContext(type=EXTENDED)` não é suportada por Web Beans simples.

## 13.2. Invocando um Web Bean a partir de um Servlet

É fácil utilizar um Web Bean a partir de um Servlet em Java EE : basta injetar o Web Bean usando a injeção de atributos ou de método de inicialização de Web Beans.

```
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");
      }
   }

}
```

O proxy cliente do Web Bean (Web Beans client proxy) cuida do encaminhamento das invocações dos métodos do Servlet para as instâncias corretas de `Credentials` e `Login` para a requisição e sessão HTTP atuais.

## 13.3. Invocando um Web Bean de um Message-Driven Bean

Injeção de Web Beans aplica-se a todos EJBs, mesmo quando esses não estão sob o controle do gerenciador do Web Bean (se tiverem sido obtidos por busca direta no JNDI, ou por injeção utilizando `@EJB`, por exemplo). Em particular, você pode usar a injeção de Web Beans em Message-Driven Beans, que não são considerados Web Beans porque você não pode injetá-los.

Você ainda pode usar bindings de interceptadores Web Beans em Message-Driven Beans.

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

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

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

}
```

Assim, receber mensagens é super fácil no ambiente Web Beans. Mas, cuidado, pois não existe um contexto de sessão ou conversação disponível quando uma mensagem é entregue a um Message-Driven Bean. Apenas `@RequestScoped` e `@ApplicationScoped` Web Beans estão disponíveis.

Também é fácil enviar mensagens usando Web Beans.

## 13.4. Endpoints JMS

O envio de mensagens usando JMS pode ser bastante complexo, devido à quantidade de objetos diferentes que precisamos utilizar. Para filas, temos `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` e `QueueSender`. Para os tópicos, temos `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` e `TopicPublisher`. Cada um desses objetos tem seu próprio ciclo de vida e modelo de threads, com que temos de nos preocupar.

A Web Beans cuida de tudo isso para nós. Tudo que precisamos fazer é declarar a fila ou o tópico no `web-beans.xml`, especificando e associando o tipo de binding e a fábrica de conexão (connection factory).

```
<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
>
```

Agora, podemos injetar a `Queue`, `QueueConnection`, `QueueSession` ou `QueueSender` para uma fila, ou `Topic`, `TopicConnection`, `TopicSession` ou `TopicPublisher` em um tópico.

```
@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) );
}
```

O ciclo de vida do objeto JMS injetado é completamente controlado pelo gerenciador do Web Bean.

## 13.5. Empacotamento e implantação

A Web Beans não define nenhum tipo especial de pacote de implantação. Você pode empacotar Web Beans em JARs, EJB-JARs ou WARs # qualquer localização de implantação do classpath da aplicação. Entretanto, cada arquivo ( JARs, EJB-JARs ou WARs) que contém Web Beans deve incluir um arquivo chamado `web-beans.xml` no diretório `META-INF` ou no diretório `WEB-INF`. O arquivo pode ser vazio. Os Web Beans implantados em pacotes que não possuem o arquivo `web-beans.xml` não estarão disponíveis para uso na aplicação.

Para execução em ambiente Java SE, Web Beans podem ser implantados em qualquer localização em que EJBs possam ser implantados para execução pelo container EJB Lite embutido (embeddable EJB Lite container). Novamente, cada localização deve conter o arquivo `web-beans.xml`.

# Estendendo a Web Beans

A Web Beans pretende ser uma plataforma para frameworks, extensões e integração com outras tecnologias. Portanto,a Web Beans expõe um conjunto de SPIs para a utilização pelos desenvolvedores de extensões portáveis para Web Beans. Por exemplo, os seguintes tipos de extensões estavam previstas pelos designers da Web Beans:

- Integração com motores de gerenciamento de processos de negócios (Business Process Management )
- integração com frameworks de terceiros, tais como Spring, Seam, GWT ou Wicket, e
- nova tecnologia baseada no modelo de programação da Web Beans.

O nervo central para estender a Web Beans é o objeto `Manager`.

## 14.1. O objeto `Manager`

A interface `Manager` permite, programaticamente, registrar e obter Web Beans, interceptadores, decoradores, observadores e contextos.

```
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);

}
```

Nós podemos obter uma instância do `Manager` via injeção:

```
@Current Manager manager
```

## 14.2. A classe `Bean`

Instâncias da classe abstrata `Bean` representam Web Beans. Existe uma instância do `Bean` registrado com o objeto `Manager` para todos os Web Beans da aplicação.

```
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);

}
```

É possível estender a classe `Bean` e registrar instâncias através da chamada `Manager.addBean()`, para fornecer suporte para novos tipos de Web Beans, além dos definidos pela especificação Web Beans (Web Beans simples e coporativos, métodos produtores e endpoints JMS). Por exemplo, poderíamos usar a classe `Bean` para permitir que os objetos gerenciados por um outro framework possam ser injetados nos Web Beans.

Existem duas subclasses de `Bean` definidas pela especificação de Web Beans: `Interceptor` e `Decorator`.

## 14.3. A interface `Context`

A interface `Context` suporta a adição de novos escopos a Web Beans, ou extensões dos escopos existentes para novos ambientes.

```
public interface Context {

   public Class<? extends Annotation> getScopeType();

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

   boolean isActive();

}
```

Por exemplo, nós poderíamos implementar `Context` para adicionar um escopo de processo de negócios a Web Beans, ou para adicionar suporte ao escopo de conversação a uma aplicação que utiliza o Wicket.

# Próximos passos

Como a Web Beans é muito nova, ainda não existe muita informação disponível online.

Claro que a especificação de Web beans é a melhor fonte para mais informações sobre Web Beans. A especificação possui cerca de 100 páginas, duas vezes o tamanho desse artigo. Mas, evidentemente, abrange muitas informações que não abordamos. A especificação está disponível em `http://jcp.org/en/jsr/detail?id=299`.

A implementação de referência de Web Beans (Web Beans RI) está sendo desenvolvida em `http://seamframework.org/WebBeans`. O blog do time de desenvolvimento da implementação de referência (RI) e do líder da especificação (Web Beans spec lead) em: `http://in.relation.to`. Esse artigo é substancialmente baseado em uma série de entradas de blogs publicados lá.

# Parte V. ReferÃªncia Ã  Web Beans

Web Beans é a implementação de referência da JSR-299 e é utilizada pelo JBoss AS e pelo Glassfish para prover serviços JSR-299 para aplicações Java Enterprise Edition. Web Beans vai além dos ambientes e APIs definidos pela especificação JSR-299 e fornece suporte a uma série de outros ambientes (tais como um servlet container como o Tomcat, ou o Java SE) e APIs e módulos adicionais (como logging, geração de XSD para os descritores de implantação XML da JSR-299).

Se pretende começar a utilizar rapidamente a Web Beans com o JBoss AS ou com o Tomcat e experimentar um dos exemplos, dê uma olhada em *Capítulo 3, Getting started with Web Beans, the Reference Implementation of JSR-299*. De qualquer maneira, continue lendo para uma discussão exaustiva da utilização da Web Beans em todos os ambientes e servidores de aplicações suportados, bem como as extensões da Web Beans .

# Servidores de Aplicação e ambientes suportados pela Web Beans

## 16.1. Utilizando a Web Beans com o JBoss AS

Além da adição de `META-INF/beans.xml` ou `WEB-INF/beans.xml`, nenhuma outra configuração especial na sua aplicação é necessária.

Se você estiver usando o JBoss AS 5.0.1.GA, então precisará instalar a Web Beans como um extra. Primeiro, precisamos dizer à Web Beans onde o JBoss está localizado. Editar o `jboss-as/build.properties` e definir a propriedade `jboss.home`. Por exemplo:

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

Agora podemos instalar a Web Beans:

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

**Nota**

Um novo deployer `webbeans.deployer` é adicionado ao JBoss AS. Isso adiciona suporte a implantações JSR-299 no JBoss AS e permite à Web Beans consultar o EJB3 container e descobrir quais EJBs estão instalados na sua aplicação.

A Web Beans está embutida em todas as versões do JBoss AS a partir da versão 5.1.

## 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

O Tomcat tem um JNDI apenas de leitura. Assim, a Web Beans não pode vincular automaticamente o Manager. Para vincular o Manager no JNDI, você deve adicionar o seguinte ao seu `META-INF/context.xml`:

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

e torná-lo disponível para a sua implantação, acrescentando-o ao `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`.

# Extensões da JSR-299 disponíveis como parte da Web Beans

**Importante**

Estes módulos são utilizáveis em qualquer implementação da JSR-299, e não apenas na 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.

# Apêndice 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. Descoberta de Web Bean (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. Serviços EJB

**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();
```

The `EjbDescriptor` is fairly self-explanatory, and should return the relevant metadata as defined in the EJB specification. In addition to these two interfaces, there is `BusinessInterfaceDescriptor` which represents a local business interface (encapsulating the interface class and jndi name used to look up an instance of the 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. Serviços JPA

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. Servicos de transação

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. O contexto de aplicação

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 e 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()`.

Para iniciar o container é chamado `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. Carregando recursos

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. O contrato com o container

There are a number of requirements that the Web Beans RI places on the container for correct functioning that fall outside implementation of APIs

Isolamento de Classloader (Classloader isolation)
: If you are integrating the Web Beans RI into an environment that supports deployment of multiple applications, you must enable, automatically, or through user configuation, classloader isolation for each Web Beans application.

Servlet
: If you are integrating the Web Beans into a Servlet environment you must register `org.jboss.webbeans.servlet.WebBeansListener` as a Servlet listener, either automatically, or through user configuration, for each Web Beans application which uses 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.

Interceptador de Session Bean (Session Bean Interceptor)

If you are integrating the Web Beans into an EJB environment you must register `org.jboss.webbeans.ejb.SessionBeanInterceptor` as a EJB interceptor for all EJBs in the application, either automatically, or through user configuration, for each Web Beans application which uses enterprise beans.

**Importante**

You must register the `SessionBeanInterceptor` as the inner most interceptor in the stack for all EJBs.

A `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`