Weld - Implementação de Referência da JSR-299

JSR-299: O novo padrão Java para injeção de dependência e gerenciamento de estado contextual

Gavin King

Pete Muir

Dan Allen

David Allen

Tradução para o Italiano: Nicola Benaglia, Francesco Milesi

Tradução para o Espanhol: Gladys Guerrero

Tradução para o Coreano: Eun-Ju Ki,

Tradução para o Chinês Tradicional: Terry Chuang

Tradução para o Chinês Simplificado: Sean Wu

Uma nota sobre nomeação e nomenclatura

Pouco antes do rascunho final da JSR-299 ser submetido, a especificação mudou seu nome de "Web Beans" para "Java Contexts and Dependency Injection for the Java EE platform", abreviado como CDI. Por um breve período após a mudança de nome, a implementação de referência adotou o nome "Web Beans". No entanto, isto acabou causando mais confusão do que resolveu e a Red Hat decidiu mudar o nome da implementação de referência para "Weld". Você ainda poderá encontrar outra documentação, blogs, postagens em fóruns, etc. que usam a nomenclatura anterior. Por favor, atualize as referências que você puder. O jogo de dar nomes acabou.

Você também descobrirá que algumas das funcionalidades que existiam na especificação agora estão ausentes, como a definição de beans em XML. Estas características estarão disponíveis como extensões portáveis para CDI no projeto Weld, e talvez em outras implementações.

Note que este guia de referência foi iniciado enquanto mudanças ainda eram realizadas na especificação. Nós fizemos o nosso melhor para atualizá-lo precisamente. Se você descobrir um conflito entre o que está escrito neste guia e a especificação, a especificação é a fonte oficial—assuma ela como correta. Se você acredita ter encontrado um erro na especificação, por favor reporte-o para o JSR-299 EG.

Parte I. Beans

A especificação *JSR-299* [http://jcp.org/en/jsr/detail?id=299] (CDI) define um conjunto de serviços complementares que ajudam a melhorar a estrutura do código da aplicação. CDI dispõe uma camada com um avançado ciclo de vida e um modelo de interação sobre os tipos de componentes Java existentes, incluindo os managed beans e Enterprise Java Beans. Os serviços da CDI fornece:

- um ciclo de vida melhorado para objetos stateful, vinculados a *contextos* bem definidos,
- uma abordagem typesafe para *injeção de dependência*,
- interação com objetos através de um *mecanismo de notificação de eventos*,
- uma melhor abordagem para associar *interceptadores* a objetos, juntamente com um novo tipo de interceptador, chamado de *decorador*, que é mais adequado para uso na resolução de problemas de negócio, e
- um *SPI* para desenvolvimento de extensões portáveis para o contêiner.

The CDI services are a core aspect of the Java EE platform and include full support for Java EE modularity and the Java EE component architecture. But the specification does not limit the use of CDI to the Java EE environment. In the Java SE environment, the services might be provided by a standalone CDI implementation like Weld (see *Seção 18.4.1, "Módulo CDI SE"*), or even by a container that also implements the subset of EJB defined for embedded usage by the EJB 3.1 specification. CDI is especially useful in the context of web application development, but the problems it solves are general development concerns and it is therefore applicable to a wide variety of application.

Um objeto vinculado a um contexto de ciclo de vida é chamado de bean. CDI inclui suporte embutido para vários tipos diferentes de bean, incluindo os seguintes tipos de componente do Java EE:

- managed beans, e
- EJB session beans.

Tanto os managed beans quanto os EJB session beans podem injetar outros beans. Mas alguns outros objetos, que não são em si beans no sentido aqui utilizado, podem também ter beans injetados via CDI. Na plataforma Java EE, os seguintes tipos de componente podem ter beans injetados:

- message-driven beans,
- interceptadores,
- servlets, filtros de servlet e escutadores de eventos servlet,
- pontos de acesso e manipuladores de serviço JAX-WS, e
- manipuladores de tag JSP e escutadores de evento em biblioteca de tag.

CDI livra o usuário de uma API desconhecida e da necessidade de reponder às seguintes questões:

- Qual é o ciclo de vida deste objeto?
- Quantos clientes simultâneos eu posso ter?
- É multithread?
- Como faço para obter acesso a ele a partir de um cliente?

- Eu preciso explicitamente destruí-lo?
- Onde devo manter referência a ele quando não estiver usando-o diretamente?
- Como posso definir uma implementação alternativa, de modo que a implementação possa variar em tempo de implantação?
- Como devo proceder no compartilhamento deste objeto com outros objetos?

CDI é mais do que um framework. É um completo e rico modelo de programação. O *tema* de CDI é *baixo acoplamento com tipagem forte*. Vamos estudar o que esta frase significa.

Um bean especifica apenas o tipo e a semântica de outros beans que ele dependa. Ele não precisa ser consciente do ciclo de vida atual, implementação concreta, modelo de threading ou outros clientes de qualquer bean que ele venha a interagir. Melhor ainda, a implementação concreta, ciclo de vida e o modelo de threading de um bean podem variar de acordo com o cenário de implantação, sem afetar qualquer cliente. Este baixo acoplamento torna seu código mais fácil de manter.

Eventos, interceptadores e decoradores melhoram o *baixo acoplamento* inerente a este 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.

O que é ainda mais poderoso (e confortável) é que CDI oferece todas essas facilidades de uma maneira *typesafe*. CDI nunca confia em identificadores baseados em strings para determinar como os objetos se relacionam. Em vez disso, CDI utiliza a informação de tipagem que já está disponível no modelo de objeto Java, aumentada com um novo padrão de programação, chamada de *anotações qualificadoras*, para unir os beans, suas dependências, seus interceptadores e decoradores, e seus consumidores de eventos. A utilização de descritores XML é minimizada para simplesmente informação específica de implantação.

Mas CDI não é um modelo de programação restritivo. Ele não diz como você deve estruturar sua aplicação em camadas, como você deve lidar com a persistência, ou qual framework web você tem que usar. Você terá que decidir esses tipos de coisas por conta própria.

CDI ainda provê um SPI abrangente, permitindo que outros tipos de objeto definidos pelas futuras especificações Java EE ou por frameworks de terceiros sejam transparentemente integrados com CDI, tirando proveito dos serviços de CDI e interagindo com qualquer outro tipo de bean.

A CDI foi influenciada por inúmeros frameworks Java existentes, incluindo Seam, Guice e Spring. Entretanto, CDI tem suas próprias e bem distintas características: mais typesafe que o Seam, mais stateful e menos centrada em XML que o Spring, mais hábil em aplicações web e corporativas que o Guice. Mas poderia ter sido nada disso sem a inspiração vinda dos frameworks mencionados e a *grande quantidade* de colaboração e trabalho duro do JSR-299 Expert Group (EG).

Finalmente, CDI é um padrão do *Java Community Process* [http://jcp.org] (JCP). Java EE 6 requer que todo servidor de aplicação compatível forneça suporte para JSR-299 (até mesmo no web profile).

Introdução

Então você está interessado em começar a escrever seu primeiro bean? Ou talvez você seja cético, imaginando que tipos de argolas a especificação CDI fará com que você atravesse! A boa notícia é que você provavelmente já escreveu e utilizou centenas, talvez milhares de beans. CDI apenas facilita a realmente utilizá-los para construir uma aplicação!

1.1. O que é um bean?

Um bean é exatamente o que você pensa que é. Só que agora ele tem uma verdadeira identidade no ambiente do contêiner.

Antes do Java EE 6, não havia uma definição clara do termo "bean" na plataforma Java EE. Claro, nós fomos chamando as classes Java usadas em aplicações web e corporativas de "beans" por anos. Houveram até um tanto de diferentes tipos de coisas chamados de "bean" em especificações EE, incluindo os beans do EJB e os managed beans do JSF. Entretanto, outros frameworks de terceiros como Spring e Seam introduziram suas próprias ideias do que significava ser um "bean". O que está faltando é uma definição comum.

Java EE 6, finalmente, estabelece que a definição comum está na especificação de Managed Beans. Managed Beans são definidos como objetos gerenciados pelo contêiner com mínimas restrições de programação, também conhecidos pelo acrônimo POJO (Plain Old Java Object). Eles suportam um pequeno conjunto de serviços básicos, como injeção de recurso, callbacks e interceptadores do ciclo de vida. Especificações complementares, tais como EJB e CDI, se estabelecem sobre este modelo básico. Porém, *afinal*, existe um conceito uniforme de um bean e um modelo de componente enxuto que está alinhado através da plataforma Java EE.

Com pouquíssimas exceções, quase toda classe Java concreta que possui um construtor com nenhum parâmetro (ou um construtor designado com a anotação `@Inject`) é um bean. Isso inclui qualquer JavaBean e qualquer EJB session bean. Se você já possui alguns JavaBeans ou session beans, eles já são beans—você não vai precisar de qualquer metadado especial adicional. Há apenas uma pequena coisa que você precisa fazer antes de começar a injetá-los dentro das coisas: você precisa colocá-los em um arquivo (um jar ou um módulo Java EE, como um war ou um jar EJB) que contenha um arquivo indicador especial: `META-INF/beans.xml`.

Os JavaBeans e EJBs que você tem escrito todo dia, até agora, não foram capazes de tirar proveito dos novos serviços definidos pela especificação CDI. Mas você será capaz de usar cada um deles com CDI—permitindo que o contêiner crie e destrua instâncias de seus beans e associando-os a um contexto designado, injetando-os dentro de outros beans, usando-os em expressões EL, especializando-os com anotações qualificadoras, até adicionando interceptadores e decoradores para eles—sem modificar seu código existente. No máximo, você precisará adicionar algumas anotações.

Agora vamos ver como criar seu primeiro bean que realmente utiliza CDI.

1.2. Arregaçando as mangas

Suponha que temos duas classes Java existentes, que estamos utilizando durante anos em várias aplicações. A primeira classe divide 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 do EJB:

```
@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 bean que faz este trabalho:

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

   @Inject
   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();
   }
}
```

Mas espere! `TextTranslator` não tem um construtor sem parâmetros! Isto é ainda um bean? Se você lembrar, uma classe que não tem um construtor sem parâmetros ainda pode ser um bean, se tiver um construtor anotado com `@Inject`.

Como você imaginou, a anotação `@Inject` tem algo a ver com injeção de dependencia! `@Inject` pode ser aplicada a um construtor ou a um método de um bean, e diz ao contêiner para chamar este construtor ou este método ao instanciar o bean. O contêiner injetará outros beans nos parâmetros do construtor ou do método.

Vamos criar um bean controlador de UI que utiliza injeção em campo para obter uma instância de `TextTranslator`, traduzindo o texto digitado por um usuário:

Vamos criar um bean controlador de UI que utiliza injeção em campo para obter uma instância de `TextTranslator`, traduzindo o texto digitado por um usuário:

```
@Named @RequestScoped
public class TranslateController {
   @Inject TextTranslator textTranslator;

   private String inputText;
   private String translation;

   // JSF action method, perhaps
   public void translate() {
      translation = textTranslator.translate(inputText);
   }

   public String getInputText() {
      return inputText;
   }

   public void setInputText(String text) {
      this.inputText = text;
   }

   public String getTranslation() {
      return translation;
   }
}
```

(1) Injeção de uma instância de `TextTranslator` em um campo

Dica

Observe que o bean controlador tem escopo de solicitação e é nomeado. Uma vez que esta combinação é tão comum em aplicações web uma anotação embutida para isto em CDI, que poderíamos ter utilizada como um atalho. Quando a anotação (estereótipo) `@Model` é declarada sobre uma classe, cria-se um bean com escopo de solicitação e nomeado.

Alternativamente, podemos obter e injetar uma instância de `TextTranslator` programaticamente a partir de uma instância de `Instance`, parametrizada com o tipo do bean:

```
@Inject Instance<TextTranslator
> textTranslatorInstance;
...
public void translate() {
   textTranslatorInstance.get().translate(inputText);
}
```

Repare que não é necessário criar um método getter ou setter para injetar um bean dentro de outro. CDI pode acessar um campo injetado diretamente (mesmo se ele for privado!), que algumas vezes ajuda a eliminar algum código supérfluo. O nome do campo é arbitrário. É o tipo do campo que determina o que é injetado.

Durante a inicialização do sistema, o contêiner deve validar que existe exatamente um bean que satisfaça cada ponto de injeção. Em nosso exemplo, se nenhuma implementação de `Translator` está disponível—se o EJB

`SentenceTranslator` não foi implantado—o contêiner iria nos informar sobre uma *dependência não satisfeita*. Se mais de uma implementação de `Translator` estivessem disponíveis, o contêiner iria nos informar sobre a *dependência ambígua*.

Antes de aprofundarmos nos detalhes, vamos fazer uma pausa e examinar a anatomia de um bean. Que aspectos do bean são significantes e o que lhe confere sua identidade? Em vez de apenas dar exemplos de beans, vamos definir o que *torna* algo um bean.

Mais sobre beans

Um bean é usualmente uma classe de aplicação que contém lógica de negócio. Pode ser chamado diretamente a partir do código Java, ou pode ser invocado por meio da Unified EL. Um bean pode acessar recursos transacionais. As dependências entre beans são gerenciadas automaticamente pelo contêiner. A maioria dos beans são *stateful* e *contextuais*. O ciclo de vida de um bean é sempre gerenciado pelo contêiner.

Vamos voltar um segundo. O que realmente significa ser *contextual*? Uma vez que os beans podem ser stateful, é importante saber *qual* instância do bean eu tenho. Ao contrário de um modelo de componentes stateless (por exemplo, stateless session beans) ou um modelo de componentes singleton (como servlets, ou singleton beans), diferentes clientes de um bean vêem o bean em diferentes estados. O estado visível ao cliente depende de para qual instância do bean o cliente tem uma referência.

No entanto, como em um modelo stateless ou singleton, mas de modo *diferente* em stateful session beans, o cliente não controla o ciclo de vida da instância pela explícita criação e destruição dela. Em vez disso, o *escopo* do bean determina:

- o ciclo de vida de cada instância do bean e
- quais clientes compartilham uma referência para uma instância específica do bean.

Para uma dada thread em uma aplicação CDI, pode haver um *contexto ativo* associado com o escopo do bean. Este contexto pode ser único para a thread (por exemplo, se o bean possui escopo de solicitação), ou pode ser compartilhado com algumas outras threads (por exemplo, se o bean possui escopo de sessão) ou mesmo com todas as outras threads (se ele possui escopo de aplicação).

Os clientes (por exemplo, outros beans) executados no mesmo contexto verão a mesma instância do bean. Mas os clientes em um diferente contexto podem ver uma instância diferente (dependendo do relacionamento entre os contextos).

Uma grande vantagem do modelo contextual é que ele permite que stateful beans sejam tratados como serviços! O cliente não precisa se preocupar com o gerenciamento do ciclo de vida do bean que ele está usando, *nem mesmo precisam saber o que é ciclo de vida.* Os beans interagem passando mensagens, e as implementações do bean definem o ciclo de vida de seu próprio estado. Os beans são de baixo acoplamento porque:

- eles interagem por meio de APIs bem definidas e públicas
- seus ciclos de vida são completamente desacoplados

We can replace one bean with another different bean that implements the same interface and has a different lifecycle (a different scope) without affecting the other bean implementation. In fact, CDI defines a simple facility for overriding bean implementations at deployment time, as we will see in *Seção 4.7, “Alternativos”*.

Note que nem todos os clientes de um bean são eles próprios também beans. Outros objetos como servlets ou message-driven beans—que são por natureza objetos não injetáveis e não contextuais —podem também obter referências para beans por meio de injeção.

2.1. A anatomia de um bean

Já chega de acenar as mãos. Mais formalmente, a anatomia de um bean, de acordo com a especificação:

> Um bean abrange os seguintes atributos:
>
> - Um conjunto (não vazio) de tipos de bean
> - Um conjunto (não vazio) de qualificadores

- Um escopo
- Opcionalmente, um nome EL do bean
- Um conjunto de vinculações com interceptadores
- Uma implementação do bean

Além disso, um bean pode ou não pode ser um bean alternativo.

Vamos ver o que toda esta nova terminologia significa.

2.1.1. Tipos e qualificadores de bean e injeção de dependência

Beans usualmente adquirem referências para outros beans por meio de injeção de dependência. Qualquer atributo injetado especifica um "contrato" que deve ser satisfeito pelo bean para ser injetado. O contrato é:

- um tipo de bean, juntamente com
- um conjunto de qualificadores.

Um tipo de bean é uma classe ou interface definida pelo usuário; um tipo que é visível ao cliente. Se o bean é um EJB session bean, o tipo do bean é a interface `@Local` ou a classe do bean da visão local. Um bean pode possuir múltiplos tipos. Por exemplo, o seguinte bean possui quatro tipos de bean:

```
public class BookShop
      extends Business
      implements Shop<Book
> {
   ...
}
```

Os tipos de bean são `BookShop`, `Business` e `Shop<Book>`, bem como o tipo implícito `java.lang.Object`. (Observe que um tipo parametrizado é um tipo de bean válido).

Entretanto, este session bean possui somente as interfaces locais `BookShop`, `Auditable` e `java.lang.Object` como tipos de bean, uma vez que a classe do bean, `BookShopBean`, não é um tipo visível ao cliente.

```
@Stateful
public class BookShopBean
      extends Business
      implements BookShop, Auditable {
   ...
}
```

Nota

The bean types of a session bean include local interfaces and the bean class local view (if any). EJB remote interfaces are not considered bean types of a session bean. You can't inject an EJB using its remote interface unless you define a *resource*, which we'll meet in *Capítulo 14, Recursos do ambiente de componentes Java EE*.

Os tipos do bean podem ser limitados a um conjunto explícito, anotando o bean com a anotação `@Typed` e listando as classes que devem ser os tipos do bean. Por exemplo, os tipos de bean desde bean foram restritos a `Shop<Book>`, juntamente com `java.lang.Object`:

```
@Typed(Shop.class)
public class BookShop
      extends Business
      implements Shop<Book
> {
   ...
}
```

Algumas vezes um tipo de bean sozinho não fornece informação suficiente para o contêiner saber qual bean injetar. Por exemplo, suponha que temos duas implementações da interface `PaymentProcessor`: `CreditCardPaymentProcessor` e `DebitPaymentProcessor`. Injetar em um campo do tipo `PaymentProcessor` introduz uma condição ambígua. Nestes casos, o cliente deve especificar algum qualidade adicional da implementação que ele está interessado. Modelamos esta categoria de "qualidade" usando um qualificador.

Um qualificador é uma anotação definida pelo usuário que é ela própria anotada com `@Qualifer`. Uma anotação de qualificador é uma extensão do sitema de tipos. Ela nos permite desambiguar um tipo sem ter que recorrer a nomes baseados em strings. Aqui está um exemplo de uma anotação de qualificador:

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

Você pode não estar acostumado a ver a definição de uma anotação. Na verdade, essa poderia ser a primeira vez que você encontrou uma. Com CDI, definições de anotação se tornará um artefato familiar conforme você for criando-os de vez em quando.

Nota

Preste atenção nos nomes das anotações embutidas em CDI e EJB. Você perceberá que elas são muitas vezes adjetivos. Nós encorajamos você a seguir esta convenção ao criar suas próprias anotações, uma vez que elas servem para descrever os comportamentos e papéis da classe.

Agora que temos definido uma anotação de qualificador, podemos utilizá-la para resolver a ambiguidade no ponto de injeção. O seguinte ponto de injeção possui o tipo de bean `PaymentProcessor` e o qualificador `@CreditCard`:

```
@Inject @CreditCard PaymentProcessor paymentProcessor
```

Para cada ponto de injeção, o contêiner pesquisa por um bean que satisfaça o contrato, um que tenha o tipo de bean e todos os qualificadores. Se ele encontrar exatamente um bean, ele injeta uma instância deste bean. Se ele não encontrar, ele reporta um erro ao usuário.

Como especificamos os qualificadores de um bean? Anotando a classe de bean, é claro! O seguinte bean possui o qualificador `@CreditCard` e implementa o tipo de bean `PaymentProcessor`. Portanto, ele satisfaz nosso ponto de injeção qualificado:

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

Nota

Se um bean ou um ponto de injeção não define explicitamente um qualificador, ele terá o qualificador padrão, `@Default`.

That's not quite the end of the story. CDI also defines a simple *resolution rule* that helps the container decide what to do if there is more than one bean that satisfies a particular contract. We'll get into the details in *Capítulo 4, Injeção e pesquisa programática de dependências*.

2.1.2. Escopo

O *escopo* de um bean define o ciclo de vida e a visibilidade de suas instâncias. O modelo de contexto da CDI é extensível, acomodando escopos arbitrários. No entanto, certos escopos importantes estão encorporados na especificação, e fornecidos pelo contêiner. Cada escopo é representado por um tipo de anotação.

Por exemplo, qualquer aplicação web pode possuir beans com *escopo de sessão*:

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

Uma instância de um bean com escopo de sessão está vinculada à sessão do usuário e é compartilhada por todas as solicitações executadas no contexto desta sessão.

Nota

Mantenha em mente que uma vez que um bean está vinculado a um contexto, ele permanece neste contexto até que o contexto seja destruído. Não existe modo algum para remover manualmente um bean daquele contexto. Se você não quer que o bean fique na sessão indefinitivamente, considere o uso de um outro escopo com um tempo de vida mais curto, como os escopos de solicitação e conversação.

Se um escopo não está explicitamente especificado, então o bean pertence a um escopo especial chamado de *pseudo-escopo dependente*. Os beans com este escopo vivem para servir o objeto no qual eles foram injetados, o que significa que seu ciclo de vida está vinculado ao ciclo de vida deste objeto.

We'll talk more about scopes in *Capítulo 5, Escopos e contextos*.

2.1.3. Nome EL

Se você quer referenciar um bean em um código não-Java que suporta expressões Unified EL, por exemplo, em uma página JSP ou JSF, você deve assinar o bean com um *nome EL*.

O nome EL é especificado usando a anotação `@Named`, como mostrado aqui:

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

Agora podemos facilmente usar o bean em qualquer página JSF ou JSP:

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

Nota

A anotação `@Named` não é o que torna a classe um bean. A maioria das classes em um arquivo de beans já são reconhecidas como beans. A anotação `@Named` apenas torna possível referenciar o bean a partir da EL, mais comumente a partir de uma visão JSF.

Nós podemos deixar o CDI escolher um nome para nós, deixando de fora o valor da anotação `@Named`:

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

O nome padrão vem do nome não-qualificado da classe, descapitalizado; neste caso, `shoppingCart`.

2.1.4. Alternativos

Nós já vimos como os qualificadores nos permite escolher entre múltiplas implementações de uma interface durante o desenvolvimento. Mas algumas vezes temos uma interface (ou outro tipo de bean), cuja implementação varia dependendo do ambiente de implantação. Por exemplo, podemos querer usar uma implementação de imitação em um ambiente de teste. Uma *alternativa* seria declarar a classe de bean com a anotação `@Alternative`.

```
public @Alternative
class MockPaymentProcessor extends PaymentProcessorImpl { ... }
```

Normalmente anotamos um bean com `@Alternative` somente quando existe alguma outra implementação de uma interface que ele implementa (ou de qualquer de seus tipos de bean). Podemos escolher entre as alternativas no momento da implantação *selecionando* uma alternativa no descritor de implantação do CDI `META-INF/beans.xml` dentro do jar ou módulo Java EE que utiliza-o. Diferentes módulos podem especificar que eles usam diferentes alternativos.

We cover alternatives in more detail in *Seção 4.7, "Alternativos"*.

2.1.5. Tipos para vinculação de interceptador

Você pode estar familiarizado com o uso de interceptadores em EJB 3.0. Em Java EE 6, esta funcionalidade foi generalizada para trabalhar com outros beans gerenciados. Está bem, você não precisa tornar seu bean um EJB apenas para interceptar seus métodos. (Berro). Então, o que CDI tem a oferecer além disso? Bem, bastante realmente. Vamos dar algumas explicações.

A maneira em que interceptadores foram definidos em Java EE 5 não foi muito intuitivo. Era necessário especificar a *implementação* do interceptador diretamente na *implementação* do EJB, seja pela anotação `@Interceptors` ou no descritor XML. Você pode muito bem apenas colocar o código do interceptador *dentro* da implementação! Em segundo lugar, a ordem na qual os interceptadores são aplicados é obtida a partir da ordem na qual eles são declarados na anotação ou no descritor XML. Talvez isto não seja tão ruim se você está aplicando os interceptadores a um único bean. Mas, se você está aplicando eles repetidamente, então há uma boa chance de você definir por descuido uma ordem diferente para diferentes beans. Agora isso é um problema.

CDI fornece uma nova abordagem para vincular interceptadores a beans que introduz um nível de indirecionamento (e, portanto, de controle). Nós temos que definir um *tipo para vinculação de interceptador* que descreve o comportamento implementado pelo interceptador.

Um tipo para vinculação de interceptador é uma anotação definida pelo usuário que é ela mesma anotada com `@InterceptorBinding`. Isto nos permite vincular as classes de interceptador a classes de bean com nenhuma dependência direta entre as duas classes.

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

O interceptador que implementa o gerenciamento de transação declara esta anotação:

```
public @Transactional @Interceptor
class TransactionInterceptor { ... }
```

Podemos aplicar o interceptador em um bean anotando a classe de bean com o mesmo tipo para vinculação de interceptador.

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

Observe que `ShoppingCart` e `TransactionInterceptor` não sabem nada sobre o outro.

Interceptadores são específicos de implantação. (Não precisamos de um `TransactionInterceptor` em nossos testes de unidade!) Por padrão, um interceptador está disabilitado. Podemos habilitar um interceptador usando o descritor de implantação CDI `META-INF/beans.xml` do jar ou módulo Java EE. Este descritor também é onde especificamos a ordem dos interceptadores.

We'll discuss interceptors, and their cousins, decorators, in *Capítulo 9, Interceptadores* and *Capítulo 10, Decoradores*.

2.2. Quais categorias de classes são beans?

Nós já vimos dois tipos de beans: JavaBeans e EJB session beans. Esta é toda a história? Na verdade, é apenas o começo. Vamos explorar as várias categorias de beans que implementações CDI devem suportar sem modificações.

2.2.1. Managed beans

Um managed bean é uma classe Java. O ciclo de vida básico e a semântica de um managed bean são definidos pelo especificação de Managed Beans. Você pode explicitamente declarar um managed bean anotando a classe do bean com `@ManagedBean`, mas em CDI você não precisa disto. De acordo com a especificação, o contêiner CDI trata qualquer classe que satisfaz as seguintes condições como um managed bean:

- Ela não é uma classe interna não-estática.
- Ela é uma classe concreta, ou é anotada com `@Decorator`.
- Ela não é anotada com uma anotação de definição de componente EJB ou declarada como classe de bean em `ejb-jar.xml`.
- Ela não implementa `javax.enterprise.inject.spi.Extension`.
- Ela tem um construtor apropriado—ou seja:
  - a classe possui um construtor sem parâmetros, ou
  - a classe declara um construtor anotado com `@Inject`.

Nota

De acordo com esta definição, entidades JPA são tecnicamente managed beans. No entanto, as entidades possuem ciclo de vida, estado e modelo de identidade próprios e especiais, e são usualmente instanciadas por JPA ou utilizando `new`. Portanto, não recomendamos injetar uma classe de entidade diretamente. Recomendamos especialmente não atribuir um escopo que não seja `@Dependent` em uma classe de entidade, uma vez que JPA não é capaz de persistir proxies injetados por CDI.

O conjunto de restrições de tipos de bean para um managed bean contém a classe do bean, qualquer superclasse e todas as interfaces que ele implementa diretamente ou indiretamente.

Se um managed bean possui um campo público, ele deve ter o escopo padrão `@Dependent`.

Managed beans suportam as chamadas `@PostConstruct` e `@PreDestroy` de seu ciclo de vida.

Session beans também são, tecnicamente, managed beans. No entanto, uma vez que eles possuem seu próprio e específico ciclo de vida e tiram vantagem de serviços corporativos adicionais, a especificação CDI considera que eles fazem parte de uma categoria diferente de beans.

2.2.2. Session beans

Session beans pertencem à especificação EJB. Eles possuem um ciclo de vida específico, gerenciamento de estado e o modelo de concorrência é diferente de outros beans geernciados e objetos Java não-gerenciados. Mas session

beans participam em CDI apenas como qualquer outro bean. Você pode injetar um session bean dentro de outro session bean, um managed bean dentro de um session bean, um session bean dentro de um managed bean, ter um managed bean observando um evento disparado por um session bean, e assim por diante.

Nota

Os massage-driven beans e entity beans são por natureza objetos não-contextuais e não podem ser injetados dentro de outros objetos. No entanto, message-driven beans podem tirar vantagem de algumas funcionadades CDI, como injeção de dependência, interceptadores e decoradores. Na verdade, CDI realizará injeção dentro de qualquer session bean ou message-driven bean, mesmo aqueles que não são instâncias contextuais.

O conjunto irrestrito de tipos de bean para um session bean contém todas as interfaces locais do bean e suas superinterfaces. Se o session bean possui uma classe de bean de visão local, o conjunto irrestrito de tipos de bean contém a classe de bean e todas as superclasses. Além disso, `java.lang.Object` é um tipo de bean de todo session bean. Porém, interfaces remotas *não* são incluídas no conjunto de tipos de bean.

Não existe razão alguma para declarar explicitamente o escopo de um stateless session bean ou singleton session bean. O contêiner EJB controla o ciclo de vida destes beans, de acordo com a semântica da declaração `@Stateless` ou `@Singleton`. Por outro lado, um stateful session bean pode possuir qualquer escopo.

Stateful session beans podem definir um *método de remoção*, anotado com `@Remove`, que é utilizado pela aplicação para indicar que uma instância deve ser destruída. No entanto, para uma instância contextual do bean—uma instância sob o controle de CDI—este método só pode ser chamado pela aplicação se o bean possuir o escopo `@Dependent`. Para beans com outros escopos, a aplicação deve deixar o contêiner destruir o bean.

Então, quando devemos usar um session bean em vez de um simples managed bean? Sempre que precisar dos serviços corporativos avançados oferecidos por EJB, tais como:

- gerenciamento de transação e segurança em nível de método,
- gerenciamento de concorrência,
- passivação em nível de instância para stateful session bean e pooling de instâncias para stateless session beans,
- invocação remota ou de serviço web, ou
- temporizadores e métodos assíncronos.

Quando não precisamos de nenhuma dessas coisas, um managed bean comum servirá muito bem.

Muitos beans (incluindo qualquer bean `@SessionScoped` ou `@ApplicationScoped`) estão disponíveis para acesso concorrente. Portanto, o gerenciamento de concorrência oferecido por EJB 3.1 é especialmente útil. A maioria dos beans com escopo de sessão e aplicação devem ser EJBs.

Os beans que mantêm referências a recursos muito pesados, ou tiram muito proveito do estado interno do avançado ciclo de vida gerenciado pelo contêiner, definido pelo modelo stateless/stateful/singleton de EJB, com seu suporte a passivação e pooling de instâncias.

Finalmente, normalmente é óbvio quando gerenciamento de transação a nível de método, segurança a nível de método, temporizadores, métodos remotos ou métodos assíncronos são utilizados.

O ponto que estamos tentando determinar é: usar um session bean quando você precisar dos serviços que ele oferece, não apenas porque você quer usar injeção de dependência, gerenciamento de ciclo de vida, ou

interceptadores. Java EE 6 fornece um modelo de programação graduado. É normalmente fácil iniciar com um managed bean habitual e, posteriormente, transformá-lo em um EJB apenas adicionando uma das seguintes anotações: `@Stateless`, `@Stateful` ou `@Singleton`.

Por outro lado, não tenha medo de usar session beans apenas porque você ouviu seus amigos dizer que eles são "pesados". Não é nada mais do que superstição pensar que alguma coisa é "mais pesada" apenas porque é hospedada nativamente dentro do contêiner Java EE, em vez de um contêiner proprietário de beans ou um framework de injeção de dependência que executa como uma camada adicional de ofuscação. E como um princípio geral, você deve desconfiar de pessoas que usam uma terminologia vagamente definida, como "pesado".

2.2.3. Métodos produtores

Nem tudo que precisa ser injetado pode ser resumido a uma classe de bean sendo instanciada pelo contêiner usando `new`. Existe uma abundância de casos onde precisamos de controle adicional. E se precisamos decidir em tempo de execução qual implementação de um dado tipo deve ser instanciado e injetado? E se precisamos injetar um objeto que é obtido ao consultar um serviço ou um recurso transacional, por exemplo, executando uma consulta JPA?

Um *método produtor* é um método que age como uma fonte de instâncias de bean. A própria declaração do método descreve o bean e o contêiner invoca o método para obter uma instância do bean quando nenhuma instância existe no contexto especificado. Um método produtor permite que a aplicação tome o controle total do processo de instanciação do bean.

Um método produtor é declarado anotando um método de uma classe de bean com a anotação `@Produces`.

```
@ApplicationScoped
public class RandomNumberGenerator {

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

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

}
```

Não podemos escrever uma classe de bean que é ela própria um número aleatório. Mas podemos certamente escrever um método que retorna um número aleatório. Ao tornar o método um método produtor, permitimos que o valor de retorno do método—neste caso um `Integer`—seja injetado. Podemos até mesmo especificar um qualificador—neste caso `@Random`, um escopo—que neste caso é por padrão `@Dependent`, e um nome EL—que neste caso é por padrão `randomNumber` de acordo com a convenção JavaBeans para nome de propriedades. Agora podemos obter um número aleatório em qualquer lugar:

```
@Inject @Random int randomNumber;
```

Até mesmo em uma expressão Unified EL:

```
<p
>Your raffle number is #{randomNumber}.</p
>
```

Um método produtor deve ser um método não-abstrato de uma classe de managed bean ou de uma classe de session bean. Um método produtor pode ser estático ou não-estático. Se o bean é um session bean, o método produtor deve ser um método de negócio do EJB ou um método estático da classe do bean.

Os tipos de bean de um método produtor depende do tipo de retorno do método:

- Se o tipo de retorno é uma interface, o conjunto ilimitado de tipos de bean contém o tipo de retorno, todas as interfaces estendidas direta ou indiretamente e `java.lang.Object`.
- Se um tipo tipo de retorno é primitivo ou é um tipo de array Java, o conjunto ilimitado de tipos de bean contém exatamente dois tipos: o tipo de retorno do método e `java.lang.Object`.
- Se o tipo de retorno é uma classe, o conjunto ilimitado de tipos de bean contém o tipo de retorno, todas superclasses e todas as interfaces implementadas direta ou indiretamente.

Nota

Os métodos e campos produtores podem ter um tipo primitivo como bean. Para o propósito de resolver dependências, tipos primitivos são considerados idênticos aos seus correspondentes tipos adaptadores em `java.lang`.

If the producer method has method parameters, the container will look for a bean that satisfies the type and qualifiers of each parameter and pass it to the method automatically—another form of dependency injection.

```
@Produces Set<Roles
> getRoles(User user) {
   return user.getRoles();
}
```

We'll talk much more about producer methods in *Capítulo 8, Métodos produtores*.

2.2.4. Campos produtores

Um *campo produtor* é uma alternativa mais simples para um método produtor. Um campo produtor é declarado ao anotar um campo de uma classe de bean com a anotação `@Produces`—a mesma anotação usada pelos métodos produtores.

```
public class Shop {
   @Produces PaymentProcessor paymentProcessor = ....;
   @Produces @Catalog List<Product
> products = ....;
}
```

As regras para determinação dos tipos de bean de um campo produtor assemelham-se às regras para métodos produtores.

A producer field is really just a shortcut that lets us avoid writing a useless getter method. However, in addition to convenience, producer fields serve a specific purpose as an adaptor for Java EE component environment injection, but to learn more about that, you'll have to wait until *Capítulo 14, Recursos do ambiente de componentes Java EE*. Because we can't wait to get to work on some examples.

Exemplo de aplicação web JSF

Vamos ilustrar essas ideias com um exemplo completo. Nós implementaremos um login/logout de usuário de uma aplicação que utiliza JSF. Primeiro, definiremos um bean com escopo de solicitação para manter o nome do usuário (username) e a senha (password) fornecidos durante o login, com as restrições definidas utilizando anotações da especificação Beans Validation:

```
@Named @RequestScoped
public class Credentials {
    private String username;
    private String password;

    @NotNull @Length(min=3, max=25)
    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    @NotNull @Length(min=6, max=20)
    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

Esse bean é vinculado ao prompt de login no seguinte formulário JSF:

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

Os usuários são representados por uma entidade JPA:

```
@Entity
public class User {
   private @NotNull @Length(min=3, max=25) @Id String username;
   private @NotNull @Length(min=6, max=20) String password;

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

```
}
```

(Observe que também vamos precisar de um arquivo `persistence.xml` para configurar a unidade de persistência JPA contendo a entidade `User`.)

O verdadeiro trabalho é realizado por um bean com escopo de sessão que mantém informações sobre o atual usuário conectado e expõe a entidade `User` para outros beans:

```
@SessionScoped @Named
public class Login implements Serializable {

   @Inject Credentials credentials;
   @Inject @UserDatabase 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);
      }
      else {
         // perhaps add code here to report a failed login
      }
   }

   public void logout() {
      user = null;
   }

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

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

}
```

`@LoggedIn` e `@UserDatabase` são anotações de qualificadores personalizados:

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

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, PARAMETER, FIELD})
public @interface UserDatabase {}
```

Precisamos de um bean adaptador para expor nossa `EntityManager` typesafe:

```
class UserDatabaseProducer {
   @Produces @UserDatabase @PersistenceContext
   static EntityManager userDatabase;
}
```

Agora, `DocumentEditor` ou qualquer outro bean, pode facilmente injetar o usuário atual:

```
public class DocumentEditor {
   @Inject Document document;
   @Inject @LoggedIn User currentUser;
   @Inject @DocumentDatabase EntityManager docDatabase;

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

Ou podemos referenciar o usuário atual em uma visão JSF:

```
<h:panelGroup rendered="#{login.loggedIn}">
   signed in as #{currentUser.username}
</h:panelGroup
>
```

Esperamos que este exemplo tenha dado um gostinho do modelo de programação em CDI. No capítulo seguinte, exploraremos a injeção de dependência com maior profundidade.

Injeção e pesquisa programática de dependências

Uma das características mais significativas do CDI—certamente a mais reconhecida—é injeção de dependência; desculpe-me, injeção de dependência com *typesafe*.

4.1. Pontos de injeção

A anotação `@Inject` nos permite definir um ponto de injeção que é injetado durante a instanciação do bean. A injeção pode ocorrer por meio de três diferentes mecanismos.

Injeção por parâmetro no *construtor do bean*:

```
public class Checkout {

   private final ShoppingCart cart;

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

}
```

Um bean pode possuir somente um construtor injetável.

Injeção por parâmetro em *método inicializador*:

```
public class Checkout {

   private ShoppingCart cart;

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

}
```

Nota

Um bean pode possuir múltiplos métodos inicializadores. Se o bean é um session bean, o método inicializador não é necessário ser um método de negócio do session bean.

E injeção direta de campos:

```
public class Checkout {

   private @Inject ShoppingCart cart;

}
```

Nota

Métodos getter e setter não são necessários para injeção em campo funcionar (exceto com managed beans do JSF).

A injeção de dependências sempre ocorre quando a instância do bean é instanciada pela primeira vez no contêiner. Simplificando um pouco, as coisas acontecem nesta ordem:

- Em primeiro lugar, o contêiner chama o construtor do bean (o construtor padrão ou um anotado com `@Inject`) para obter uma instância do bean.
- Em seguida, o contêiner inicializa os valores de todos os campos injetados do bean.
- Em seguida, o contêiner chama todos os métodos inicializadores do bean (a ordem de chamada não é portável, não confie nela).
- Finalmente, o método `@PostConstruct`, se for o caso, é chamado.

(A única complicação é que o contêiner pode chamar métodos inicializadores declarados por uma superclasse antes de inicializar campos injetados declarados por uma subclasse.)

Nota

Uma grande vantagem de injeção em construtores é que isto nos permite que o bean seja imutável.

CDI também suporta injeção de parâmetro para alguns outros métodos que são invocados pelo contêiner. Por exemplo, a injeção de parâmetro é suportada em métodos produtores:

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

This is a case where the `@Inject` annotation *is not* required at the injection point. The same is true for observer methods (which we'll meet in *Capítulo 11, Eventos*) and disposer methods.

4.2. Como as injeções são obtidas

A especificação CDI define um procedimento, chamado de *resolução segura de tipos*, que o contêiner segue ao indentificar o bean a ser injetado em um ponto de injeção. Este algoritmo parece complexo no início, mas uma vez quq você o entende, é realmente muito intuitivo. A resolução segura de tipos é realizada durante a inicialização do sistema, o que significa que o contêiner informará ao desenvolvedor imediatamente se alguma das dependências de um bean não puder ser satisfeita.

O objetivo deste algoritmo é permitir que múltiplos beans implementem o mesmo tipo de bean e também:

- permitir que o cliente escolha qual implementação ele necessita utilizando um *qualificador* ou
- permitir ao implantador (deployer) da aplicação escolher qual implentação é adequada para uma determinada implantação, sem alterações para o cliente, ao ativar ou desativar um *alternativo*, ou
- permitir que os beans sejam isolados dentro de módulos separados.

Obviamente, se você possui exatamente um bean de um dado tipo, e um ponto de injeção com este mesmo tipo, então o bean A irá para onde pedir um A. Este é o cenário mais simples possível. Quando você começar sua aplicação, você terá provavelmente vários desses.

Mas então, as coisas começam a ficar complicadas. Vamos explorar como o contêiner determina qual bean injetar em casos mais avançados. Nós iniciaremos dando um olhar mais atento em qualificadores.

4.3. Anotações de qualificadores

Se temos mais do que um bean que implementa um tipo de bean específico, o ponto de injeção pode especificar exatamente qual bean deve ser injetado usando uma anotação de qualificador. Por exemplo, pode haver duas implementações de `PaymentProcessor`:

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Onde `@Synchronous` e `@Asynchronous` são anotações de qualificadores:

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

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

Um desenvolvedor de um bean cliente utiliza a anotação de qualificador para especificar exatamente qual bean deve ser injetado.

Utilizando injeção por campos (field injection):

```
@Inject @Synchronous PaymentProcessor syncPaymentProcessor;
@Inject @Asynchronous PaymentProcessor asyncPaymentProcessor;
```

Utilizando injeção de método de inicialização:

```
@Inject
public void setPaymentProcessors(@Synchronous PaymentProcessor syncPaymentProcessor,
                                 @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Usando injeção no construtor:

```
@Inject
public Checkout(@Synchronous PaymentProcessor syncPaymentProcessor,
                @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Anotações de qualificadores podem também qualificar argumentos de métodos produtores, destruidores ou observadores. Combinar argumentos qualificados com métodos produtores é uma boa forma para ter uma implementação de um tipo de bean selecionado em tempo de execução com base no estado do sistema:

```
@Produces
PaymentProcessor getPaymentProcessor(@Synchronous PaymentProcessor syncPaymentProcessor,
                                     @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   return isSynchronous() ? syncPaymentProcessor : asyncPaymentProcessor;
}
```

Se um campo injetado ou um parâmetro de um construtor de bean ou método inicializador não é explicitamente anotado com um qualificador, o qualificador padrão, `@Default`, é assumido.

Agora, você pode estar pensando, *"Qual a diferença entre usar um qualificador e apenas especificar a exata classe de implementação que você deseja?"* É importante entender que um qualificador é como uma extensão da interface. Ele não cria uma dependência direta para qualquer implementação em particular. Podem existir várias implementações alternativas de `@Asynchronous PaymentProcessor`!

4.4. Os qualificadores embutidos `@Default` e `@Any`

Sempre que um bean ou ponto de injeção não declara explicitamente um qualificador, o contêiner assume o qualificador `@Default`. Em algum momento, você precisará declarar um ponto de injeção sem especificar um qualificador. Existe um qualificador para isso também. Todos os beans possuem o qualificador `@Any`. Portanto, ao especificar explicitamente `@Any` em um ponto de injeção, você suprime o qualificador padrão, sem restringir os beans que são elegíveis para injeção.

Isto é especialmente útil se você quiser iterar sobre todos os beans com um certo tipo de bean. Por exemplo:

```
@Inject
void initServices(@Any Instance<Service
> services) {
   for (Service service: services) {
      service.init();
   }
}
```

4.5. Qualificadores com membros

As anotações Java podem possuir membros. Podemos usar membros de anotação para discriminar melhor um qualificador. Isso impede uma potencial explosão de novas anotações. Por exemplo, em vez de criar vários qualificadores representando diferentes métodos de pagamento, podemos agregá-los em uma única anotação com um membro:

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

Então selecionamos um dos possíveis valores do membro ao aplicar o qualificador:

```
private @Inject @PayBy(CHECK) PaymentProcessor checkPayment;
```

Podemos forçar o contêiner a ignorar um membro de um tipo de qualificador ao anotar o membro com `@Nonbinding`.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
   PaymentMethod value();
   @Nonbinding String comment() default "";
}
```

4.6. Múltiplos qualificadores

Um ponto de injeção pode especificar múltiplos qualificadores:

```
@Inject @Synchronous @Reliable PaymentProcessor syncPaymentProcessor;
```

Neste caso, somente um bean que possua *ambas* anotações de qualificador seriam elegíveis para injeção.

```
@Synchronous @Reliable
public class SynchronousReliablePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

4.7. Alternativos

Os alternativos são beans cuja implementação é específica para um módulo cliente ou cenário de implantação específico. Este alternativo define uma implementação simulada de `@Synchronous PaymentProcessor` e `@Asynchronous PaymentProcessor`, tudo em um:

```
@Alternative @Synchronous @Asynchronous
public class MockPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Por padrão, os beans com `@Alternative` estão desabilitados. Precisamos *habilitar* um alternativo no descritor `beans.xml` de um arquivo de bean para torná-lo disponível para instanciação e injeção. Esta ativação somente se aplica aos beans neste arquivo.

```
<beans
   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/beans_1_0.xsd">
   <alternatives>
         <class
>org.mycompany.mock.MockPaymentProcessor</class>
   </alternatives>
</beans
>
```

Quando uma dependência ambígua houver em um ponto de injeção, o contêiner tenta resolver a ambiguidade procurando por um bean alternativo habilitado entre os beans que podem ser injetados. Se existir exatamente um alternativo habilitado, este será o bean a ser injetado.

4.8. Corrigindo dependências não satisfeitas e ambíguas

O algoritmo de resolução segura de tipos falha quando, após considerar as anotações de qualificadores em todos os beans que implementam o tipo de bean de um ponto de injeção e filtrar os beans desabilitados (beans com `@Alternative` que não estão explicitamente habilitados), o contêiner não é capaz de identificar exatamente um bean para injetar. O contêiner abortará a implantação, nos informando sobre as dependências não satisfeitas ou ambíguas.

Durante o progresso de seu desenvolvimento, você vai encontrar essa situação. Vamos aprender como resolvê-la.

Para corrigir uma *dependência não satisfeita*:

- crie um bean que implemente o tipo de bean e possua todos os tipos de qualificador do ponto de injeção,
- certifique-se que o bean que você já possui esteja no classpath do módulo com o ponto de injeção, ou
- habilite explicitamente um bean `@Alternative` que implemente o tipo de bean e possua os tipos de qualificador apropriados, usando `beans.xml`.

Para corrigir uma *dependência ambígua*:

- introduza um qualificador para distinguir entre as duas implementações do tipo de bean,
- desabilite um dos beans anotando-o com `@Alternative`,
- mova uma das implementações para um módulo que não está no classpath do módulo com o ponto de injeção, ou
- desabilite um dos beans `@Alternative` que estão tentando ocupar o mesmo espaço, usando `beans.xml`.

Veja *este FAQ* [http://sfwk.org/Documentation/HowDoAResolveAnAmbiguousResolutionExceptionBetweenAProducerMethodAndARawType] para instruções passo-a-passo de como resolver uma exceção de resolução ambígua entre um tipo de bean e um método produtor que retorna o mesmo tipo de bean.

Apenas lembre-se: "Só pode haver um."

On the other hand, if you really do have an optional or multivalued injection point, you should change the type of your injection point to `Instance`, as we'll see in *Seção 4.10, “Obtendo uma instância contextual através de pesquisa programática”*.

Agora há mais uma questão que você precisa estar ciente quando usar o serviço de injeção de dependência.

4.9. Proxies clientes

Os clientes de um bean injetado não costumam manter uma referência direta para uma instância do bean, a menos que o bean seja um objeto dependente (com escopo `@Dependent`).

Imagine que um bean vinculado ao escopo da aplicação mantenha uma referência direta para um bean vinculado ao escopo da solicitação. O bean com escopo de aplicação é compatilhado entre várias solicitações diferentes. No entanto, cada solicitação deverá ver uma instância diferente do bean com escopo de solicitação—a atual!

Agora imagine que um bean vinculado ao escopo da sessão mantenha uma referência direta para um bean vinculado ao escopo da aplicação. Em algum momento, o contexto da sessão é serializado para o disco, a fim de usar a memória de forma mais eficiente. No entanto, a instância do bean com escopo de aplicação não deve ser serializado junto com o bean de escopo de sessão! Ele pode obter esta referência a qualquer momento. Não há necessidade de armazená-lo!

Portanto, a menos que um bean possua o escopo padrão `@Dependent`, o contêiner deve injetar indiretamente todas as referências para o bean através de um objeto proxy. Este *proxy cliente* é responsável por assegurar que a instância do bean que recebe uma invocação de método seja a instância que está associada ao contexto atual. O proxy cliente também permite que beans vinculados a contextos, como o contexto de sessão, sejam serializados para o disco sem serializar recursivamente outros beans injetados.

Infelizmente, devido às limitações da linguagem Java, alguns tipos Java não podem ser feitos proxies pelo contêiner. Se um ponto de injeção declarado com um destes tipos referencia um bean com qualquer escopo diferente de `@Dependent`, o contêiner abortará a implantação, nos informando sobre o problema.

Os seguintes tipos Java não podem ser "proxied" pelo contêiner:

- classes que não possuem um construtor não privado sem parâmetros, e
- classes que são declaradas `final` ou que tenham um método `final`,

- arrays e tipos primitivos.

Geralmente é muito fácil de corrigir um problema de dependência com proxies. Se um ponto de injeção do tipo X resulta em uma dependência que não pode ser feito um proxy, simplesmente:

- adicione um construtor sem parâmetros em X,
- modifique o tipo do ponto de injeção para `Instance<X>`,
- introduza uma interface Y, implementada pelo bean injetado, e mude o tipo do ponto de injeção para Y, ou
- se tudo isso falhar, mude o escopo do bean a injetar para `@Dependent`.

Nota

Uma versão futura do Weld provavelmente suportará uma solução não padrão para esta limitação, usando APIs não portáveis da JVM:

- Sun, IcedTea, Mac: `Unsafe.allocateInstance()` (A mais eficiente)
- IBM, JRockit: `ReflectionFactory.newConstructorForSerialization()`

Mas não somos obrigados a implementar isto ainda.

4.10. Obtendo uma instância contextual através de pesquisa programática

Em certas situações, injeção não é o meio mais conveniente de obter uma referência contextual. Por exemplo, não pode ser usada quando:

- o tipo do bean ou qualificadores variam dinamicamente em tempo de execução, ou
- dependendo da implantação, pode haver nenhum bean que satisfaça o tipo e qualificadores, ou
- gostaríamos de realizar uma iteração sobre todos os beans de um certo tipo.

Nestas situações, a aplicação pode obter uma instância da interface `Instance`, parametrizada para o tipo do bean, por injeção:

```
@Inject Instance<PaymentProcessor
> paymentProcessorSource;
```

O método `get()` de `Instance` produz uma instância contextual do bean.

```
PaymentProcessor p = paymentProcessorSource.get();
```

Qualificadores podem ser especificados em uma de duas maneiras:

- anotando o ponto de injeção `Instance`, ou
- passando qualificadores para o método `select()` de `Event`.

Especificar os qualificadores no ponto de injeção é muito, muito mais fácil:

```
@Inject @Asynchronous Instance<PaymentProcessor
> paymentProcessorSource;
```

Agora, o `PaymentProcessor` retornado por `get()` terá o qualificador `@Asynchronous`.

Alternativamente, podemos especificar o qualificador dinamicamente. Primeiro, adicionamos o qualificador `@Any` no ponto de injeção, para suprimir o qualificador padrão. (Todos os beans possuem o qualificador `@Any`.)

```
@Inject @Any Instance<PaymentProcessor
> paymentProcessorSource;
```

Em seguida, precisamos obter uma instância de nosso tipo de qualificador. Uma vez que anotações são interfaces, não podemos apenas escrever `new Asynchronous()`. Também é bastante tedioso criar uma implementação concreta de um tipo de anotação a partir do zero. Em vez disso, o CDI nos permite obter uma instância do qualificador criando uma subclasse da classe auxiliar `AnnotationLiteral`.

```
abstract class AsynchronousQualifier
extends AnnotationLiteral<Asynchronous
> implements Asynchronous {}
```

E alguns casos, podemos utilizar uma classe anônima:

```
PaymentProcessor p = paymentProcessorSource
   .select(new AnnotationLiteral<Asynchronous
>() {});
```

No entanto, não podemos utilizar uma classe anônima para implementar um tipo de qualificador com membros.

Agora, finalmente, podemos passar o qualificador para o método `select()` de `Instance`.

```
Annotation qualifier = synchronously ?
      new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor p = anyPaymentProcessor.select(qualifier).get().process(payment);
```

4.11. O objeto `InjectionPoint`

Existem certos tipos de objetos dependentes (beans com escopo `@Dependent`) que precisam saber alguma coisa sobre o objeto ou ponto de injeção no qual eles são injetados para serem capazes de fazer o que fazem. Por exemplo:

- A categoria de log para um `Logger` depende da classe do objeto que a possui.
- A injeção do valor de um parâmetro ou cabeçalho HTTP depende de qual nome de parâmetro ou cabeçalho foi especificado no ponto de injeção.

- Injeção do resultado da avaliação de uma expressão EL depende da expressão que foi especificada no ponto de injeção.

Um bean com escopo `@Dependent` pode injetar uma instância de `InjectionPoint` e acessar metadados relacionados com o ponto de injeção ao qual ele pertence.

Vejamos um exemplo. O seguinte código é prolixo e vulnerável a problemas de refatoração:

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

Este método produtor pouco inteligente lhe permite injetar um `Logger` da JDK sem especificar explicitamente a categoria de log:

```
class LogFactory {

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

}
```

Podemos agora escrever:

```
@Inject Logger log;
```

Não está convencido? Então aqui está um segundo exemplo. Para injetar parâmetros HTTP, precisamos definir um tipo de qualificador:

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

Gostaríamos de usar este tipo de qualificador em pontos de injeção do seguinte modo:

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

O seguinte método produtor faz o trabalho:

```
class HttpParams

   @Produces @HttpParam("")
```

```
   String getParamValue(InjectionPoint ip) {
      ServletRequest request = (ServletRequest) FacesContext.getCurrentInstance().getExternalContext().getRequest
      return request.getParameter(ip.getAnnotated().getAnnotation(HttpParam.class).value());
   }

}
```

Observe que a aquisição da solicitação neste exemplo é centrada em JSF. Para uma solução mais genérica você pode escrever seu próprio produtor para a solicitação e a injetar como um parâmetro do método.

Observe que o membro `value()` da anotação `HttpParam` é ignorado pelo contêiner uma vez que ele está anotado com `@Nonbinding.`

O contêiner fornece um bean embutido que implementa a interface `InjectionPoint`:

```
public interface InjectionPoint {
   public Type getType();
   public Set<Annotation
> getQualifiers();
   public Bean<?> getBean();
   public Member getMember();
   public Annotated getAnnotated();
   public boolean isDelegate();
   public boolean isTransient();
}
```

Escopos e contextos

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

- Quando uma nova instância de qualquer bean com esse escopo é criada
- Quando uma instância existente de qualquer bean com esse escopo é destruída
- Quais referências injetadas referem-se a qualquer instância de um bean com esse escopo

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

Nota

Entidades JPA não se encaixam muito bem nesse modelo. Entidades possuem seu próprio ciclo de vida e modelo de identidade que não pode ser mapeado adequadamente para o modelo utilizado em CDI. Portanto, não recomendamos o tratamento de entidades como beans CDI. Você certamente vai ter problemas se tentar dar a uma entidade um escopo diferente do escopo padrão `@Dependent`. O proxy cliente irá atrapalhar se você tentar passar uma instância injetada para o `EntityManager` do JPA.

5.1. Tipos de escopo

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

```
@ScopeType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
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. Você pode esperar uma implementação do escopo de negócio, por exemplo, em uma versão futura do Seam.

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

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

Normalmente, você usará um dos escopos pré-definidos do CDI.

5.2. Escopos pré-definidos

CDI possui quatro escopos pré-definidos:

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

Para uma aplicação web que utiliza CDI:

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

Uma extensão do CDI pode implementar o suporte para o escopo de conversação em outros frameworks web.

Os escopos de solicitação e aplicação também estão disponíveis:

- durante invocações de métodos remotos de EJB,
- durante invocações de métodos assíncronos de EJB,
- durante timeouts de EJB,
- durante a entrega de mensagem a um message-driven bean,
- durante a entrega de mensagem a um `MessageListener`, e
- durante a invocação de um web service

Se a aplicação tentar invocar um bean com um escopo que não possui um contexto ativo, uma `ContextNotActiveException` é lançada pelo contêiner em tempo de execução.

Os managed beans com escopo `@SessionScoped` ou `@ConversationScoped` devem ser serializáveis, uma vez que o contêiner mantém a sessão HTTP resguardada ao longo do tempo.

Três dos quatro escopos pré-definidos devem ser extremamente familiares a todos os desenvolvedores Java EE, então não vamos perder tempo discutindo-os aqui. No entanto, um dos escopos é novo.

5.3. O escopo de conversação

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

- é demarcado explicitamente pela aplicação, e
- mantém estado associado a uma aba específica do navegador web em uma aplicação JSF (navegadores tendem a compartilhar os cookies do domínio, que também inclui o cookie de sessão, entre as abas, de modo que este não é o caso para o escopo de sessão).

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

O contexto de conversação fica ativo durante qualquer solicitação JSF. A maioria das conversações é destruída no final da solicitação. Se uma conversação deve manter estado através de múltiplas solicitações, ela deve ser explicitamente promovida a uma *conversação de longa duração*.

5.3.1. Demarcação de contexto

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

```
@Inject Conversation conversation;
```

Para promover a conversação associada com a solicitação 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 solicitação atual, chame `end()`.

No exemplo a seguir, um bean com escopo de conversação controla a conversação na qual estiver associado:

```
@ConversationScoped @Stateful
public class OrderBuilder {
   private Order order;
   private @Inject Conversation conversation;
   private @PersistenceContext(type = EXTENDED) EntityManager em;

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

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

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

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

   @Remove
   public void destroy() {}
}
```

Este bean é capaz de controlar seu próprio ciclo de vida através do uso da API `Conversation`. Mas alguns outros beans possuem um cliclo vida que depende totalmente de um outro objeto.

5.3.2. Propagação de conversação

O contexto de conversação propaga-se automaticamente em qualquer solicitação JSF (formulário de submissão JSF) ou redirecionamento. E não se propaga automaticamente em requisições não JSF, por exemplo, navegação através de um link.

Nós podemos forçar a propagação da conversação em uma solicitação não JSF incluindo o identificador único da conversação como um parâmetro da solicitação. A especificação CDI reserva o parâmetro denominado `cid`

para essa utilização. O identificador único da conversação pode ser obtido a partir do objeto `Conversation`, que possui o nome de bean `conversation` em EL.

Portanto, o seguinte link propaga a conversação:

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

É provavelmente melhor usar um dos componentes de link em JSF 2:

```
<h:link outcome="/addProduct.xhtml" value="Add Product">
   <f:param name="cid" value="#{javax.enterprise.context.conversation.id}"/>
</h:link
>
```

Dica

O contexto da conversação se propaga sobre redirecionamentos, tornando muito fácil implementar o padrão POST-then-redirect, sem ter de recorrer a construções frágeis, como um objeto "flash". O contêiner adiciona automaticamente o identificador da conversação na URL redirecionada como um parâmetro de solicitação.

5.3.3. Tempo limite de conversação

O contêiner pode destruir uma conversação e todo estado mantido em seu contexto, a qualquer momento, a fim de preservar recursos. A implementação do CDI irá fazer isso normalmente, com base em algum tipo de tempo limite (timeout)—embora isso não seja exigido pela especificação. O tempo limite é o período de inatividade antes que a conversação seja destruída (em contraste com a quantidade de tempo que a conversação está ativa).

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

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

5.4. O pseudo-escopo singleton

Além dos quatro escopos pré-definidos, CDI também suporta dois *pseudo-escopos*. O primeiro é o *pseudo-escopo singleton*, que especificamos usando a anotação `@Singleton`.

Nota

Ao contrário dos outros escopos, os quais pertencem ao pacote `javax.enterprise.context`, a anotação `@Singleton` é definida no pacote `javax.inject`.

Você pode adivinhar o que "singleton" significa aqui. Ele significa que um bean é instanciado apenas uma vez. Infelizmente, existe um pequeno problema com este pseudo-escopo. Os beans com escopo `@Singleton` não possuem um objeto de proxy. Os clientes mantêm uma referência direta para a instância singleton. Portanto, precisamos considerar o caso de um cliente que pode ser serializado, por exemplo, qualquer bean com escopo `@SessionScoped` ou `@ConversationScoped`, qualquer objeto dependente de um bean com escopo `@SessionScoped` ou `@ConversationScoped`, ou qualquer bean de sessão com estado.

Agora, se a instância singleton é simples objeto imutável e serializável como uma string, um número ou uma data, provavelmente não importaremos muito se ele ficar duplicado na serialização. No entanto, isso faz com que ele deixe de ser um verdadeiro singleton, e podemos muito bem apenas o declarar com o escopo padrão.

Existem várias maneiras para garantir que o bean singleton permaneça como singleton quando seu cliente se torna serializável.

- mandar o bean singleton implementar `writeResolve()` e `readReplace()` (como definido pela especificação de serialização Java),
- certificar-se que o cliente mantém apenas uma referência transiente para o bean singleton, ou
- dar ao cliente uma referência do tipo `Instance<X>`, onde X é o tipo do bean singleton.

Uma quarta e melhor solução é usar `@ApplicationScoped`, permitindo que o contêiner faça um proxy do bean, e resolva automaticamente os problemas de serialização.

5.5. O pseudo-escopo dependente

Finalmente, CDI possui o assim chamado *pseudo-escopo dependente*. Esse é o escopo padrão para um bean que não declare explicitamente um tipo de escopo.

Por exemplo, esse bean possui o tipo de escopo `@Dependent`:

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

Uma instância de um bean dependente nunca é compartilhada entre clientes diferentes ou pontos de injeção diferentes. Ele é estritamente um *objeto dependente* de algum outro objeto. Ele é instanciado quando o objeto a que ele pertence é criado, e destruído quando o objeto a que ele pertence é destruído.

Se uma expressão Unified EL se refere a um bean dependente pelo seu nome EL, uma instância do bean é instaciada toda vez que a expressão é avaliada. A instância não é reutilizada durante qualquer outra avaliação de expressão.

Nota

Se você precisa acessar um bean diretamente pelo nome EL em uma página JSF, você provavelmente precisa dar a ele um escopo diferente de `@Dependent`. Caso contrário, qualquer valor que for definido no bean por uma entrada JSF será perdido imediatamente. É por isso que CDI possui o estereótipo `@Model`; ele permite que você dê um nome ao bean e define seu escopo para `@RequestScoped` de uma só vez. Se você precisa acessar um bean que realmente *necessita* possuir o escopo `@Dependent` a partir de uma página JSF, injete-o dentro de um bean diferente e exponha-o em EL por meio de um método getter.

Os beans com escopo `@Dependent` não precisam de um objeto de proxy. O cliente mantém uma referência direta para sua instância.

CDI torna fácil a obtenção de uma instância dependente de um bean, mesmo se o bean já tiver declarado como um bean com algum outro tipo de escopo.

5.6. O qualificador `@New`

O qualificador pré-definido `@New` nos permite obter um objeto dependente de uma classe especificada.

```
@Inject @New Calculator calculator;
```

A classe deve ser um managed bean ou session bean válido, mas não precisa ser um bean habilitado.

Isso funciona 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 {
   @Inject Calculator calculator;
   @Inject @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`.

This feature is particularly useful with producer methods, as we'll see in *Capítulo 8, Métodos produtores*.

Parte II. Primeiros Passos com Weld, a Implementação de Referência de CDI

Weld, a Implementação de Referência (RI) da JSR-299, está sendo desenvolvido como parte do *projeto Seam* [http://seamframework.org/Weld]. Você pode baixar a mais recente versão pública do Weld na *página de download* [http://seamframework.org/Download]. Informações sobre o repositório de código fonte do Weld e instruções sobre como obter e compilar o código podem ser encontradas na mesmo página.

Weld provê um completo SPI permitindo que contêineres Java EE como JBoss AS e GlassFish usem Weld como sua implementação CDI embutida. Weld também roda em mecanismos servlet como Tomcat e Jetty, ou mesmo em um bom ambiente Java SE.

Weld vem com uma extensiva biblioteca de exemplos, os quais são um grande ponto de partida para aprender CDI.

Iniciando com o Weld

O Weld vem com vários exemplos. Nós recomendamos que você inicie com `examples/jsf/numberguess` e `examples/jsf/translator`. O numberguess é um exemplo web (war) contendo somente managed beans não-transacionais. Este exemplo pode ser executado em uma ampla variedade de servidores, incluindo JBoss AS, GlassFish, Apache Tomcat, Jetty, Google App Engine, e qualquer contêiner Java EE 6 compatível. O translator é um exemplo corporativo (ear) que contém sessions beans. Este exemplo deve executar no JBoss AS 6.0, GlassFish 3.0 ou qualquer contêiner Java EE 6 compatível.

Ambos exemplos usam JSF 2.0 como framework web e podem ser encontrados no diretório `examples/jsf` da distribuição do Weld.

6.1. Pré-requisitos

Para executar os exemplos com os scripts de construção fornecidos, você precisará dos seguintes itens:

- a última versão do Weld, a qual contém os exemplos
- Ant 1.7.0, para construir e implantar os exemplos
- um ambiente de execução suportado (versões mínimas requeridas)
  - JBoss AS 6.0.0,
  - GlassFish 3.0,
  - Apache Tomcat 6.0.x (somente com o exemplo war), ou
  - Jetty 6.1.x (somente com o exemplo war)
- (opcionalmente) Maven 2.x, para executar os exemplos em um contêiner servlet embutido

Nota

Você precisará de uma instalação completa do Ant 1.7.0. Algumas distribuições Linux fornecem apenas uma instalação parcial do Ant que faz com que a construção falhe. Se você tiver problemas, verifique se ant-nodeps.jar está no classpath.

Nas próximas seções, você usará o comando do Ant (`ant`) para invocar o script de construção do Ant em cada exemplo para compilar, montar e implantar o exemplo no JBoss AS e, para o exemplo war, no Apache Tomcat. Você também pode implantar o artefato produzido (war ou ear) em qualquer outro contêiner que suporte Java EE 6, como o GlassFish 3.

Se você tem o Maven instalado, você pode usar o comando do Maven (`mvn`) para compilar e montar o artefato autônomo (war ou ear) e, para o exemplo war, executá-lo em um contêiner embutido.

As seções abaixo cobrem os passos para implantar com Ant e Maven em detalhes. Vamos iniciar com o JBoss AS.

6.2. Implantando no JBoss AS

Para implantar os exemplos no JBoss AS, você precisará do *JBoss AS 6.0.0* [http://jboss.org/jbossas/] ou acima. Se uma versão da linha JBoss AS 6.0 ainda não estiver disponível, você pode baixar um *nightly snapshot*

[http://hudson.jboss.org/hudson/view/JBoss%20AS/job/JBoss-AS-6.0.x/]. A razão do JBoss AS 6.0.0 ou acima ser requerido é porque esta é a primeira versão que possui suporte embutido a CDI e Bean Validation, tornando-o próximo suficiente a Java EE 6 para executar os exemplos. A boa notícia é que não existem modificações adicionais que você tenha que fazer no servidor. Está pronto para funcionar!

Depois de ter baixado o JBoss AS, extrai-o. (Recomendamos renomear a pasta para incluir o qualificador `as`, assim fica claro que é o servidor de aplicação). Você pode mover a pasta extraída para qualquer lugar que você queira. Seja qual for o local, este é o que chamaremos de diretório de instalação do JBoss AS, ou `JBOSS_HOME`.

```
$
> unzip jboss-6.0.*.zip
$
> mv jboss-6.0.*/ jboss-as-6.0
```

Para que os scripts de construção saibam onde implantar o exemplo, você tem que dizer a eles onde encontrar sua instalação do JBoss AS (ou seja, `JBOSS_HOME`). Crie um novo arquivo com o nome `local.build.properties` no diretório dos exemplos do Weld obtido e atribua o caminho até sua instalação do JBoss AS para a propriedade `jboss.home`, como a seguir:

```
jboss.home=/path/to/jboss-as-6.0
```

Agora você está pronto para implantar seu primeiro exemplo!

Vá para o diretório `examples/jsf/numberguess` e execute o alvo `deploy` no Ant:

```
$
> cd examples/jsf/numberguess
$
> ant deploy
```

Se você já não estiver, inicie o JBoss AS. Você pode também iniciar o JBoss AS a partir de um shell Linux:

```
$
> cd /path/to/jboss-as-6.0
$
> ./bin/run.sh
```

uma janela de comandos do Windows:

```
$
> cd c:\path\to\jboss-as-6.0\bin
$
> run
```

ou você pode iniciar o servidor usando uma IDE, como o Eclipse.

Nota

Se você estiver usando o Eclipse, deve considerar seriamente instalar os add-ons do *JBoss Tools* [http://www.jboss.org/tools], que inclui uma ampla variedade de ferramentas para desenvolvimento com JSR-299 e Java EE, bem como uma visão melhorada para o servidor JBoss AS.

Aguarde uns poucos segundos para a aplicação ser implantada (ou o servidor de aplicação iniciar) e veja se você pode determinar a abordagem mais eficiente para apontar o número aleatório na URL local *http://localhost:8080/weld-numberguess*.

Nota

O script de construção do Ant inclui alvos adicionais para o JBoss AS implantar e desimplantar o arquivo no formato explodido ou empacotado e deixar tudo certo.

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

O segundo exemplo de partida, `weld-translator`, traduzirá seu texto para Latin. (Bem, não realmente, mas o mínimo está lá para você ao menos começar. Boa sorte!) Para testá-lo, mude para o diretório do exemplo de tradutor e execute o alvo deploy:

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

Nota

O tradutor utiliza session beans, os quais estão empacotados em um módulo EJB dentro de um ear. Java EE 6 permitirá que session beans sejam implantados em módulos war, mas isto é um tópico para um capítulo posterior.

Novamente, aguarde uns poucos segundos para a aplicação ser implantada (se você está realmente entediado, leia as mensagens de log), e visite *http://localhost:8080/weld-translator* para começar a pseudo-tradução.

6.3. Implantando no GlassFish

Implantar no GlassFish deve ser fácil e familiar, certo? Afinal, é a implementação de referência da Java EE 6 e Weld é a implementação de referência da JSR-299, o que significa que o Weld vem integrado ao GlassFish. Então, sim, é tudo muito fácil e familiar.

Para implantar os exemplos no GlassFish, você precisará o versão final do *GlassFish V3* [https://glassfish.dev.java.net/downloads/v3-final.html]. Selecione a versão que termina com `-unix.sh` ou `-windows.exe` de acordo com sua plataforma. Após a conclusão do download, execute o instalador. Em Linux/Unix, primeiramente, você precisará tornar o script executável.

```
$
> chmod 755 glassfish-v3-unix.sh
$
> ./glassfish-v3-unix.sh
```

No Windows você pode simplesmente clicar sobre o executável. Siga as instruções no instalador. Ele criará um único domínio nomeado como `domain1`. Você usará este domínio para implantar os exemplos. Nós recomendamos que você escolha `7070` como a porta HTTP principal para evitar conflitos com uma instância em execução do JBoss AS (ou Apache Tomcat).

Se você tiver implantado um dos exemplos de partida, `weld-numberguess` ou `weld-translator`, no JBoss AS, então você já possui o artefato implantável que necessita. Se não, mude para qualquer um dos dois diretórios e mande construí-lo.

```
$
> cd examples/jsf/numberguess (or examples/jsf/translator)
$
> ant package
```

O artefato implantável para o exemplo `weld-numberguess`, com o nome `weld-numberguess.war`, é criado no diretório `target` do exemplo. O arquivo para o exemplo `weld-translator`, com o nome `weld-translator.ear`, é criado no diretório `ear/target` do exemplo. Tudo que você precisa fazer agora é implantá-lo no GlassFish.

Um modo para implantar aplicações no GlassFish é utilizando o *GlassFish Admin Console* [http://localhost:4848]. Para ter o Admin Console em executação, você precisa iniciar um domínio do GlassFish, em nosso caso `domain1`. Mude para a pasta `bin` no diretório onde você instalou o GlassFish e execute o seguinte comando:

```
$
> asadmin start-domain domain1
```

Após uns poucos segundos você pode visitar o Admin Console no navegador através da URL *http://localhost:4848*. Na árvore do lado esquerdo da página, clique em "Aplicações", depois no botão "Implantar..." abaixo do título "Aplicações" e selecione o artefato implantável para qualquer um dos dois exemplos. O implantador deve reconhecer que você selecionou um artefato Java EE e permite que você o inicie. Você pode ver os exemplos rodando em *http://localhost:7070/weld-numberguess* ou *http://localhost:7070/weld-translator*, dependendo de qual exemplo você implantou.

Alternativamente, você pode implantar a aplicação no GlassFish utilizando o comando asadmin:

```
$
> asadmin deploy target/weld-numberguess.war
```

A razão pela qual o mesmo artefato pode ser implantado tanto no JBoss AS quanto no GlassFish, sem qualquer modificação, é porque todas as funcionalidades utilizadas fazem parte da plataforma padrão. E o que uma plataforma competente é capaz de fazer!

6.4. Implantando no Apache Tomcat

Os contêineres servlet não são requeridos para suportar serviços Java EE como CDI. No entanto, você pode usar CDI em um contêiner servlet, como o Tomcat, incorporando uma implementação standalone de CDI, como o Weld.

O Weld vem com um servlet listener que inicializa o ambiente CDI, registra o `BeanManager` no JNDI e oferece injeção dentro de servlets. Basicamente, ele simula um pouco do trabalho realizado pelo contêiner Java EE. (Mas você não terá funcionalidades corporativas como session beans e transações gerenciadas pelo contêiner.)

Vamos colocar a extensão servlet do Weld para rodar no Apache Tomcat. Primeiro você precisará baixar o Tomcat 6.0.18 ou posterior em *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], e o descompactar.

```
$
> unzip apache-tomcat-6.0.18.zip
```

Você possui duas opções para implantar a aplicação no Tomcat. Você pode implantá-la publicando o artefato no diretório de implantação automática usando o Ant ou você pode implantar no servidor via HTTP usando um plugin do Maven. A abordagem do Ant não requer que você tenha o Maven instalado, por isso começaremos por aí. Se você quer usar o Maven, você pode simplesmente pular esta seção.

6.4.1. Implantando com o Ant

Para que o Ant coloque o artefato no diretório de implantação automática do Tomcat, ele precisa saber onde a instalação do Tomcat está localizada. Novamente, precisamos definir uma propriedade no arquivo `local.build.properties` dentro do diretório de exemplos do Weld obtido. Se você ainda não tiver criado este arquivo, faça isso agora. Em seguida, atribua o caminho de sua instalação do Tomcat na propriedade `tomcat.home`.

```
tomcat.home=/path/to/apache-tomcat-6
```

Agora você está pronto para implantar o exemplo numberguess no Tomcat!

Mude para o diretório `examples/jsf/numberguess` novamente e execute o alvo `deploy` no Ant para o Tomcat:

```
$
> cd examples/jsf/numberguess
$
> ant tomcat.deploy
```

Nota

O script de contrução do Ant inclui alvos adicionais para o Tomcat implantar e desimplantar o arquivo no formato explodido ou empacotado. Eles possuem os mesmos nomes usados nos alvos do JBoss AS, prefixados com "tomcat.".

- `ant tomcat.restart` - implanta o exemplo no formato explodido no Tomcat
- `ant tomcat.explode` - atualiza o exemplo explodido, sem reiniciar a implantação
- `ant tomcat.deploy` - implanta o exemplo no formato jar compactado no Tomcat
- `ant tomcat.undeploy` - remove o exemplo do Tomcat

Se você já não estiver, inicie o Tomcat. Você pode também iniciar o Tomcat a partir de um shell Linux:

```
$
> cd /path/to/apache-tomcat-6
$
> ./bin/start.sh
```

uma janela de comandos do Windows:

```
$
> cd c:\path\to\apache-tomcat-6\bin
$
> start
```

ou você pode iniciar o servidor usando uma IDE, como o Eclipse.

Aguarde uns poucos segundos para a aplicação ser implantada (ou o servidor de aplicação iniciar) e veja se você pode calcular a abordagem mais eficiente para apontar o número aleatório na URL local *http://localhost:8080/weld-numberguess*!

6.4.2. Implantando com o Maven

Você também pode implantar a aplicação no Tomcat usando o Maven. Esta seção é um pouco mais avançada, portanto pule-a, a não ser que você esteja ansioso para usar o Maven nativamente. Claro, primeiro você precisará ter certeza que possui o Maven instalado em seu caminho, similar à forma que você configurou o Ant.

O plugin do Maven comunica com o Tomcat via HTTP, de modo que não importa onde você instalou o Tomcat. No entanto, a configuração do plugin assume que você está rodando o Tomcat com sua configuração padrão, como localhost e na porta `8080`. O arquivo `readme.txt` no diretório do exemplo possui informações sobre como modificar as configurações do Maven para acomodar uma configuração diferente.

Para permitir que o Maven comunique com o Tomcat via HTTP, edite o arquivo `conf/tomcat-users.xml` em sua instalação do Tomcat e adicione a seguinte linha:

```
<user username="admin" password="" roles="manager"/>
```

Reinicie o Tomcat. Agora você pode implantar a aplicação no Tomcat com o Maven usando este comando:

```
$
> mvn compile war:exploded tomcat:exploded -Ptomcat
```

Uma vez que a aplicação está implantada, você pode reimplantá-la usando este comando:

```
$
> mvn tomcat:redeploy -Ptomcat
```

O argumento `-Ptomcat` ativa o profile `tomcat` definido no Maven POM (`pom.xml`). Entre outras coisas, este profile ativa o plugin do Tomcat.

Ao invés de enviar o contêiner para uma instalação standalone do Tomcat, você também pode executar a aplicação em um contêiner embutido do Tomcat 6:

```
$
> mvn war:inplace tomcat:run -Ptomcat
```

A vantagem de utilizar o servidor embutido é que as mudanças nos ativos em `src/main/webapp` entrar em vigor imediatamente. Se uma mudança em um arquivo de configuração da webapp é realizada, a aplicação pode automaticamente ser reimplantada (dependendo da configuração do plugin). Se você fizer uma alteração em um recurso do classpath, você precisa executar uma recompilação:

```
$
> mvn compile war:inplace -Ptomcat
```

Existem muitos outros goals do Maven que você pode usar se estiver dissecando o exemplo, o qual está documentado no arquivo `readme.txt` do exemplo.

6.5. Implantando no Jetty

O suporte ao Jetty nos exemplos é uma adição mais recente. Já que o Jetty é tradicionalmente usado com o Maven, não existem alvos do Ant para ele. Você deve invocar o Maven diretamente para implantar os exemplos no Jetty sem fazer mais nada. Além disso, somente o exemplo `weld-numberguess` está configurado para suportar o Jetty neste momento.

Se você leu toda a seção do Tomcat até o fim, então você está pronto para seguir. A execução do Maven assemelha-se à implantação embutida do Tomcat. Se não, não fique preocupado. Nós ainda iremos abordar tudo que você precisa saber novamente nesta seção.

O Maven POM (`pom.xml`) inclui um profile nomeado `jetty` que ativa o plugin Maven Jetty, que você pode usar para iniciar o Jetty no modo embutido e implantar a aplicação devidamente. Você não precisa de mais nada instalado, exceto ter o comando do Maven (`mvn`) em seu caminho. O restante será baixado da internet quando a construção for executada.

Para rodar o exemplo `weld-numberguess` no Jetty, vá para o diretório do exemplo e execute o goal `inplace` do plugin Maven war seguido pelo goal `run` do plugin Maven Jetty com o profile `jetty` habilitado, como a seguir:

```
$
> cd examples/jsf/numberguess
$
> mvn war:inplace jetty:run -Pjetty
```

A saída do log do Jetty será mostrada no console. Uma vez que o Jetty tenha reportado a implantação da aplicação, você pode acessá-la através da seguinte URL: *http://localhost:9090/weld-numberguess*. A porta está definida nas configurações do plugin Maven Jetty dentro do profile `jetty`.

Qualquer mudança nos recursos em `src/main/webapp` entra em vigor imediatamente. Se uma mudança em um arquivo de configuração da webapp é realizada, a aplicação pode ser reimplantada automaticamente. O comportamento de reimplantação pode ser sintonizado na configuração do plugin. Se você fizer uma mudança em um recurso do, você precisará executar um build e o goal `inplace` do plugin Maven war, novamente com o profile `jetty` habilitado.

```
$
> mvn compile war:inplace -Pjetty
```

O goal `war:inplace` copia as classes e jars compilados dentro de `src/main/webapp`, nos diretórios `WEB-INF/classes` e `WEB-INF/lib`, respectivamente, misturando arquivos fontes e compilados. No entanto, o build resolve estes arquivos temporários excluindo-os do war empacotado ou limpando-os durante a fase clean do Maven.

Você tem duas opções se quiser rodar o exemplo no Jetty a partir da IDE. Você pode instalar o plugin *m2eclispe* [http://m2eclipse.sonatype.org/] e executar os goals como descrito acima. Sua outra opção é iniciar o contêiner Jetty a partir de uma aplicação Java.

Primeiro, inicialize o projeto Eclipse:

```
$
> mvn clean eclipse:clean eclipse:eclipse -Pjetty-ide
```

Em seguida, monte todos os recursos necessários em `src/main/webapp`:

```
$
> mvn war:inplace -Pjetty-ide
```

Agora você está pronto para rodar o servidor no Eclipse. Importe o projeto para seu workspace do Eclipse usando "Import Existing Project into Workspace". Então, encontre a start class em `src/jetty/java` e execute seu método main como uma Java Application. O Jetty será executado. Você pode ver a aplicação através da seguinte URL: *http://localhost:8080*. Preste atenção especial na porta presente na URL e a ausência de um caminho de contexto à direita.

Agora que você possui as aplicações de partida implantadas no servidor de sua escolha, provavelmente vai querer saber um pouco sobre como eles realmente funcionam.

Mergulhando nos exemplos do Weld

É hora de tirar as tampas e mergulhar dentro das aplicações de exemplo do Weld. Vamos começar com o mais simples dos dois exemplos, `weld-numberguess`.

7.1. O exemplo numberguess em detalhes

Na aplicação numberguess você tem 10 tentativas para adivinhar um número entre 1 e 100. Depois de cada tentativa, você é informado se seu palpite foi muito alto ou muito baixo.

O exemplo numberguess é composto de uma série de beans, arquivos de configuração e páginas em Facelets (JSF), empacotados como um módulo war. Vamos começar examinando os arquivos de configuração.

Todos os arquivos de configuração para este exemplo estão localizados no `WEB-INF/`, o qual pode ser encontrado no diretório `src/main/webapp` do exemplo. Primeiro, nós temos a versão JSF 2.0 de `faces-config.xml`. Uma versão padronizada de Facelets é o view handler padrão em JSF 2.0, então não há realmente nada a ser configurado. Assim, a configuração consiste apenas ao elemento raíz.

```
<faces-config version="2.0"
   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_2_0.xsd">
</faces-config
>
```

Existe também um arquivo `beans.xml` vazio, que diz ao contêiner para procurar por beans nesta aplicação e ativar os serviços CDI.

Finalmente, temos o familiar `web.xml`:

```
<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
>weld-jsf-numberguess-war</display-name>
   <description                                                    (1)
>Weld JSF numberguess example (war)</description>

   <servlet>
      <servlet-name
>Faces Servlet</servlet-name>
      <servlet-class                                              (2)
>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup
>1</load-on-startup>
   </servlet>
```

```
                                                                    3
   <servlet-mapping>
      <servlet-name
>Faces Servlet</servlet-name>
      <url-pattern
>*.jsf</url-pattern>                                                 4
   </servlet-mapping>

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

   <session-config>
      <session-timeout
>10</session-timeout>
   </session-config>

</web-app
>
```

1. Enable and initialize the JSF servlet
2. Configure requests for URLs ending in `.jsf` to be handled by JSF
3. Tell JSF that we will be giving our JSF views (Facelets templates) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Nota

This demo uses JSF 2 as the view framework, but you can use Weld with any servlet-based web framework, such as JSF 1.2 or Wicket.

Let's take a look at the main JSF view, `src/main/webapp/home.xhtml`.

```
<!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">

   <ui:composition template="/template.xhtml">                      1
      <ui:define name="content">
         <h1
>Guess a number...</h1>
         <h:form id="numberGuess">                                  2
            <div style="color: red">
               <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.biggest}.
            You have #{game.remainingGuesses} guesses remaining.
         </div>

         <div>
            Your guess:
            <h:inputText id="inputGuess" value="#{game.guess}"
               size="3" required="true" disabled="#{game.number eq game.guess}"
               validator="#{game.validateNumberRange}"/>
            <h:commandButton id="guessButton" 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 the built-in templating language for JSF. Here we are wrapping our page in a template which defines the layout.
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 sentence changes to make sure they know the number range of a valid guess.
4. This input field is bound to a bean property using a value expression.
5. A validator binding 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 bounds 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 bean.

O exemplo consiste em 4 classes, as duas primeiras são qualificadores. Primeiramente temos o qualificador `@Random`, usado para injetar um número aleatório:

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

Existe também o qualificador `@MaxNumber`, usado para injetar o número máximo que pode ser gerado:

```
@Qualifier
@Target( { TYPE, METHOD, PARAMETER, FIELD })
```

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

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

```
@ApplicationScoped
public class Generator implements Serializable {

   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;
   }

}
```

O bean `Generator` possui escopo de aplicação, assim não obtemos uma instância de Random diferente a cada vez.

Nota

A declaração de pacote e as importações foram removidas destas listagens. A listagem completa está disponível no código fonte do exemplo.

O último bean na aplicação é a classe `Game` com escopo de sessão. Este é o principal ponto de entrada da aplicação. É responsável por criar ou redefinir o jogo, capturando e validando o palpite do usuário e fornecendo resposta ao usuário com uma `FacesMessage`. Nós utilizamos o método de pós-construção para inicializar o jogo recuperando um número aleatório a partir do bean `@Random Instance<Integer>`.

Você notará que também adicionamos a anotação `@Named` nesta classe. Esta anotação somente é necessária quando você quer tornar o bean acessível em uma página JSF por meio de EL (ou seja, #{game}).

```
@Named
@SessionScoped
public class Game implements Serializable {

   private int number;
   private int guess;
   private int smallest;
   private int biggest;
```

```
   private int remainingGuesses;

   @Inject @MaxNumber private int maxNumber;
   @Inject @Random Instance<Integer
> randomNumber;

   public Game() {}

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

   @PostConstruct
   public void reset() {
      this.smallest = 0;
      this.guess = 0;
      this.remainingGuesses = 10;
      this.biggest = maxNumber;
      this.number = randomNumber.get();
   }

  public void validateNumberRange(FacesContext context,  UIComponent toValidate, Object value) {
      if (remainingGuesses <= 0) {
         FacesMessage message = new FacesMessage("No guesses left!");
         context.addMessage(toValidate.getClientId(context), message);
         ((UIInput) toValidate).setValid(false);
         return;
      }
      int input = (Integer) value;

      if (input < smallest || input
> biggest) {
         ((UIInput) toValidate).setValid(false);

         FacesMessage message = new FacesMessage("Invalid guess");
         context.addMessage(toValidate.getClientId(context), message);
      }
   }

   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;
   }

}
```

7.1.1. O exemplo numberguess no Apache Tomcat ou Jetty

Uma série de modificações devem ser feitas no artefato numberguess para que seja implatado no Tomcat ou Jetty. Primeiro, o Weld deve ser implantado como uma biblioteca de aplicação Web em `WEB-INF/lib` já que o contêiner servlet não oferece os serviços de CDI. Para sua conveniência, oferecemos um único jar suficiente para executar o Weld em qualquer contêiner servlet (incluindo o Jetty), `weld-servlet.jar`.

Nota

Você deve também incluir os jars para JSF, EL e as anotações comuns (`jsr250-api.jar`), todos aqueles que são fornecidos pela plataforma Java EE (um servidor de aplicação Java EE). Você está começando a entender porque uma plataforma Java EE é importante?

Segundo, precisamos especificar explicitamente o servlet listener no `web.xml`, porque o contêiner não fará isto por você. O servlet listener inicia o Weld e controla sua interação com as solicitações.

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

Quando o Weld inicia, ele coloca o `javax.enterprise.inject.spi.BeanManager`, o SPI portável para obtenção de instâncias de bean, no ServletContext com um nome igual ao nome da interface totalmente qualificada. Normalmente você não precisa acessar esta interface, mas o Weld faz uso dela.

7.2. O exemplo numberguess para Java SE com Swing

Este exemplo mostra como utilizar a extensão Weld SE em uma aplicação Java SE feita em Swing com nenhum EJB ou dependências com servlet. Este exemplo pode ser encontrado na pasta `examples/se/numberguess` da distribuição padrão do Weld.

7.2.1. Criando o projeto no Eclipse

Para usar o exemplo numberguess com Weld SE no Eclipse, você pode abrir o exemplo normalmente utilizando o *plugin m2eclipse* [http://m2eclipse.sonatype.org/].

If you have m2eclipse installed, you can open any Maven project directly. From within Eclipse, select *File -> Import... -> Maven Projects*. Then, browse to the location of the Weld SE numberguess example. You should see that Eclipse recognizes the existence of a Maven project.

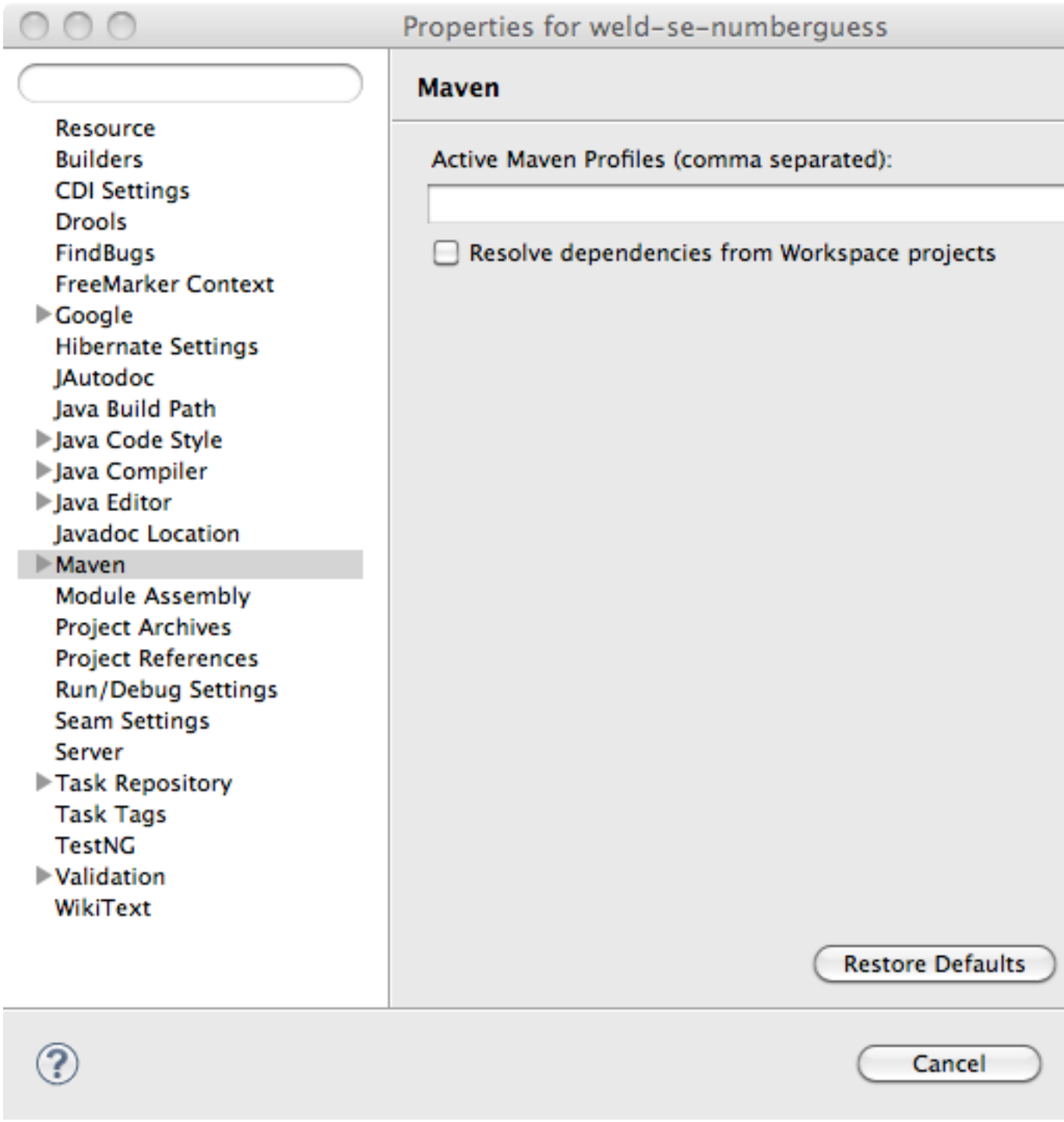
Isto criará um projeto em seu workspace com o nome `weld-se-numberguess`.

Se você não está usando o plugin m2eclipse, você tem que seguir diferentes passos para importar o projeto. Primeiro, entre no exemplo numberguess do Weld em SE, então execute o plugin Maven Eclipse com o profile jetty ativado, conforme a seguir:

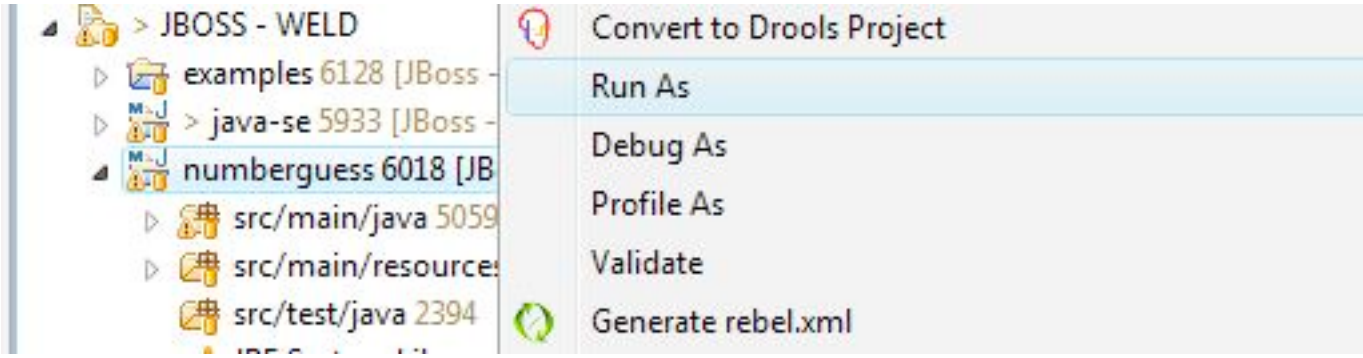
É hora de ver o exemplo rodando!

7.2.2. Rodando o exemplo dentro do Eclipse

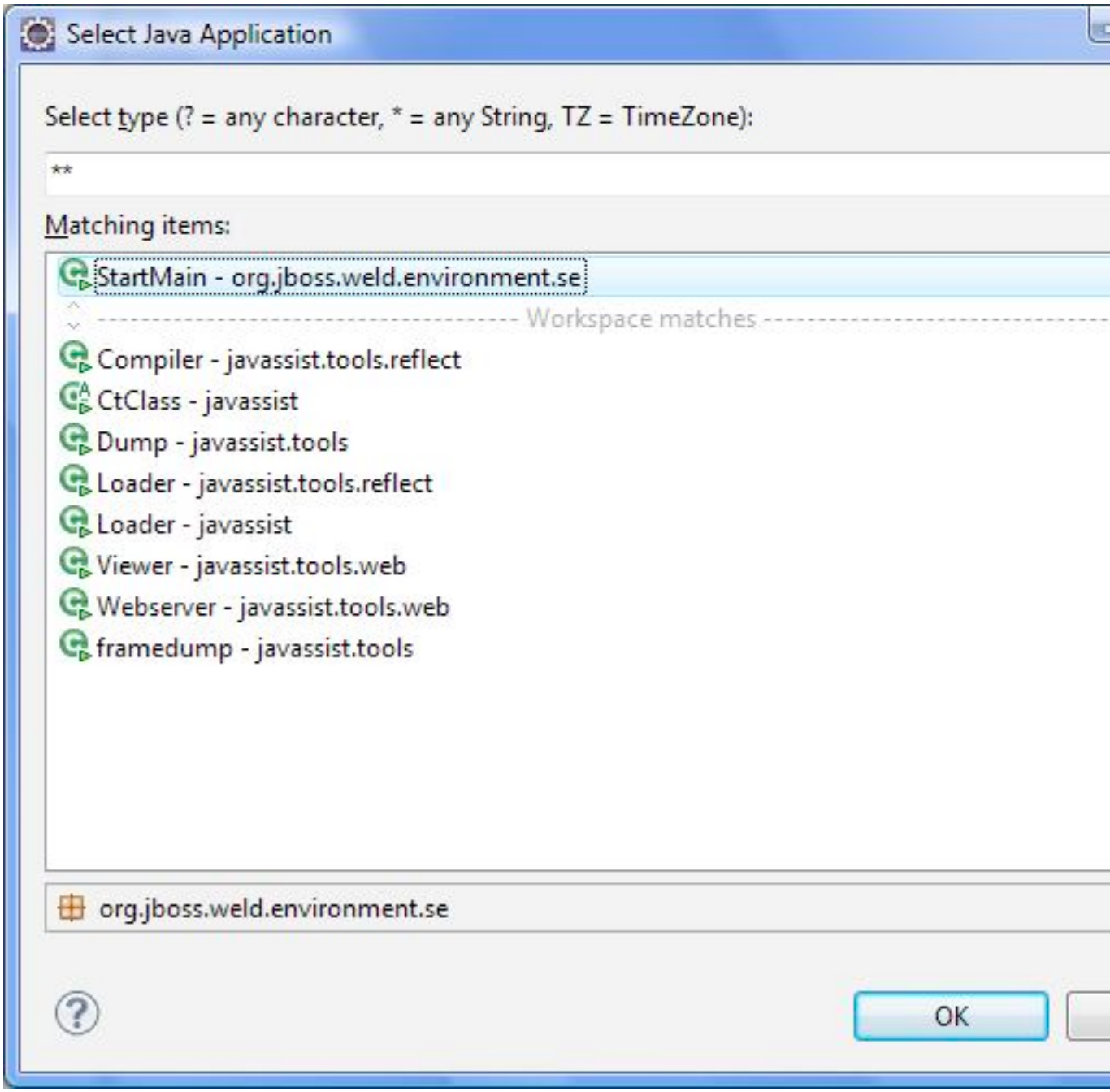
Disable m2eclipse's *Workspace Resolution*, to make sure that Eclipse can find `StartMain`. Right click on the project, and choose *Properties -> Maven*, and uncheck *Resolve dependencies from Workspace projects*:



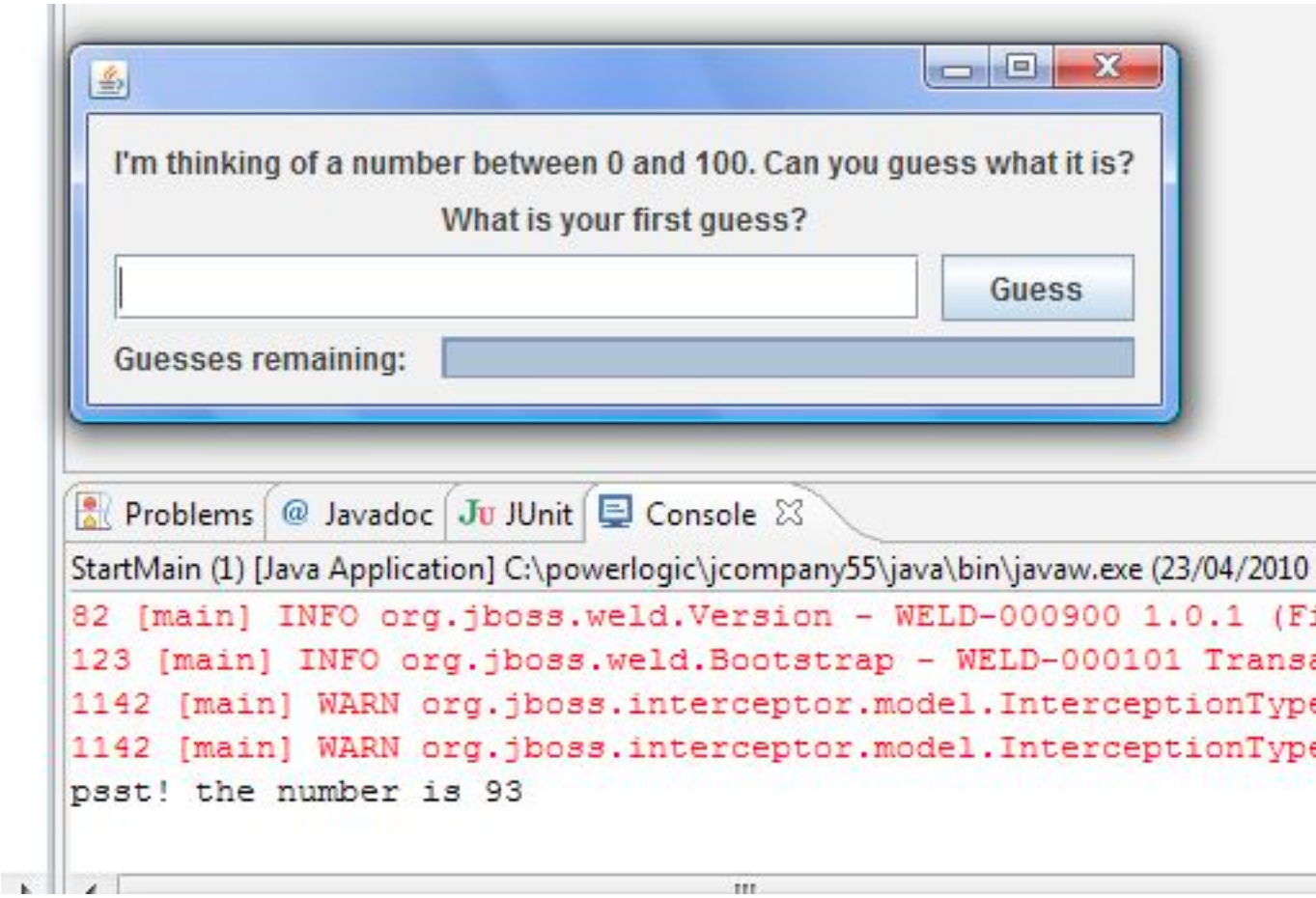
Right click on the project, and choose *Run As -> Run As Application*:



Localize a classe `StartMain`:



Agora a aplicação deve iniciar!



7.2.3. Rodando o exemplo a partir da linha de comando

- Garanta que o Maven 3 esteja instalado e presente em seu PATH
- Garanta que a variável de ambiente JAVA_HOME esteja apontando para sua instalação do JDK
- Abra a linha de comando ou janela de terminal no diretório `examples/se/numberguess`
- Execute o seguinte comando

```
mvn -Drun
```

7.2.4. Entendendo o código

Vamos dar uma olhada no código e arquivos de configuração interessantes que compõem este exemplo.

Como usual, existe um arquivo `beans.xml` vazio no pacote raíz (`src/main/resources/beans.xml`), o qual marca esta aplicação como uma aplicação CDI.

A lógica principal do jogo está em `Game.java`. Aqui está o código para esta classe, destacando os pontos que diferem da versão web:

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

   public static final int MAX_NUM_GUESSES = 10;

   private Integer number;
   private int guess = 0;
   private int smallest = 0;

   @Inject
   @MaxNumber
   private int maxNumber;

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

   @Inject
   Generator rndGenerator;

   public Game()
   {
   }

   ...                                                                (3)

   public boolean isValidNumberRange()
   {
      return validNumberRange;
   }

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

   public boolean isGameLost()
   {
      return guess != number && remainingGuesses <= 0;
   }                                                                  (4)

   public boolean check()
   {
      boolean result = false;

      if (checkNewNumberRangeIsValid())
      {
         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()
   {
      this.smallest = 0;
      this.guess = 0;
      this.remainingGuesses = 10;
      this.biggest = maxNumber;
      this.number = rndGenerator.next();
   }
}
```

(5)

1. The bean is application scoped rather than session scoped, since an instance of a Swing application typically represents a single 'session'.
2. Notice that the bean is not named, since it doesn't need to be accessed via EL.
3. In Java SE there is no JSF `FacesContext` to which messages can be added. 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. Since there is no dedicated validation phase, validation of user input is performed during the `check()` method.
5. The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. Note that it can't use `Instance.get()` like the JSF example does because there will not be any active contexts like there are 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
{
```

```
   @Inject                                                                    (1)
   private Game game;

   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 guessed 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
{
   @Inject                                                          (1)
   private Game game;

   @Inject                                                          (2)
   private MessageGenerator msgGenerator;

   public void start(@Observes ContainerInitialized event)          (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)
   {
      game.reset();                                                 (6)
      refreshUI();
   }

   private void refreshUI() {
      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 prescribed Weld SE way, by observing the `ContainerInitialized` event.
4. This method initializes all of the Swing components. Note the use of the `msgGenerator` here.
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 messages 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.

7.3. O exemplo translator em detalhe

O exemplo translator pegará qualquer sentença que voc# entrar e as traduzirá para Latim. (Bem, não realmente, mas a base está aí para você implementar. Boa sorte!)

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

Nota

Java EE 6, que vem com EJB 3.1, permite que você empacote EJBs em um war, o que tornará sua estrutura muito mais simples! Todavia, existem outras vantagens em usar um ear.

Primeiro, vamos dar uma olhada no agregador eear, que está localizado no diretório `ear` do exemplo. O Maven automaticamente gera o `application.xml` para nós com esta configuração de plugin:

```
<plugin>
   <groupId
>org.apache.maven.plugins</groupId>
   <artifactId
>maven-ear-plugin</artifactId>
   <configuration>
      <modules>
```

```
         <webModule>
            <groupId
>org.jboss.weld.examples.jsf.translator</groupId>
            <artifactId
>weld-jsf-translator-war</artifactId>
            <contextRoot
>/weld-translator</contextRoot>
         </webModule>
      </modules>
   </configuration>
</plugin
>
```

Esta configuração sobrescreve o caminho do contexto web, resultando nesta URL para a aplicação: *http://localhost:8080/weld-translator*.

Nota

Se você não estiver utilizando o Maven para gerar estes arquivos, você deve precisar de um `META-INF/application.xml`:

```
<application version="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/application_5.xsd">

  <display-name
>weld-jsf-translator-ear</display-name>
  <description
>The Weld JSF translator example (ear)</description>

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

Agora, vamos dar uma olhada no war, que está localizado no diretório `war` do exemplo. Da mesma forma do exemplo numberguess, temos um `faces-config.xml` para JSF 2.0 e um `web.xml` (para ativar JSF), ambos dentro de `src/main/webapp/WEB-INF`.

O mais interessante é a visão JSF usada para traduzir texto. Como no exemplo numberguess, possuímos um template (aqui omitido por brevidade) que circunda o formulário:

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

   <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 algum texto no textarea à esquerda, e pressionar o botão translate para ver o resultado à direita.

Finalmente vamos dar uma olhada no módulo EJB, o qual está localizado no diretório `ejb` do exemplo. Em `src/main/resources/META-INF` existe apenas um `beans.xml` vazio, usado para indicar que o jar possui beans.

Nós temos deixado a parte mais interessante por último, o código! O projeto possui dois simples beans, `SentenceParser` e `TextTranslator`, e dois session beans, `TranslatorControllerBean` e `SentenceTranslator`. Você já deve estar bem familiarizado com beans agora, assim destacamos apenas as partes mais interessantes.

Tanto `SentenceParser` quanto `TextTranslator` são beans dependentes, e `TextTranslator` utiliza injeção no construtor:

```
public class TextTranslator implements Serializable {

   private SentenceParser sentenceParser;

   @EJB private Translator translator;

   @Inject public TextTranslator(SentenceParser sentenceParser) {
      this.sentenceParser = sentenceParser;
   }
```

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

}
```

`TextTranslator` usa o bean `SentenceParser` (realmente apenas uma simples classe Java!) para analisar a sentença e então chama o stateless bean com a interface local de negócio `Translator` para realizar a tradução. É onde a mágica acontece. Certamente, nós não pudemos desenvolver um tradutor completo, mas é suficientemente convincente para quem não conhece Latim!

```
@Stateless
public class SentenceTranslator implements Translator {

   public String translate(String sentence) {
      return "Lorem ipsum dolor sit amet";
   }

}
```

Finalmente, existe um controlador orientado na interface com o usuário. Este é um stateful session bean com escopo de requisição e nomeado, o qual injeta o tradutor. Ele coleta o texto do usuário e o despacha para o tradutor. O bean também possui getters e setters para todos os campos na página.

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

   @Inject private TextTranslator translator;

   private String inputText;

   private String translatedText;

   public void translate() {
      translatedText = translator.translate(inputText);
   }

   public String getText() {
      return inputText;
   }

   public void setText(String text) {
      this.inputText = text;
   }

   public String getTranslatedText() {
      return translatedText;
```

```
   }

   @Remove public void remove() {}

}
```

Isto conclui nosso rápido passeio com os exemplos de partida do Weld. Para mais informações sobre o Weld, por favor visite *http://www.seamframework.org/Weld*.

Parte III. Baixo aclopamento com tipificação forte

O primeiro grande tema de CDI é *baixo acoplamento*. Já vimos três meios de alcançar baixo acoplamento:

- *alternativos* permitem polimorfismo durante a implantação,
- *métodos produtores* permitem polimorfismo durante a execução, e
- *gerenciamento contextual do ciclo de vida* desacopla os ciclos de vida dos beans.

Estas técnicas permitem uma baixa acoplagem do cliente e servidor. O cliente não é mais fortemente vinculado a uma implementação de uma interface, nem é necessário gerenciar o ciclo de vida da implementação. Esta abordagem deixa que *objetos com estado interajam como se eles fossem serviços*.

Baixo acoplamento torna um sistema mais *dinâmico*. O sistema pode responder a mudanças de uma maneira bem definida. No passado, os frameworks que tentaram fornecer as facilidades listadas acima, invariavelmente fizeram sacrificando a tipagem segura (mais notavelmente pelo uso de descritores XML). CDI é a primeira tecnologia, e certamente a primeira especificação na plataforma Java EE, que alcança este nível de baixo acoplamento de uma maneira typesafe.

CDI fornece três importantes facilidades extras adicionalmente ao objetivo da baixa acoplagem:

- *interceptadores* desacoplam questões técnicas da lógica de negócio,
- *decoradores* podem ser usados para desacoplar algumas questões de negócio, e
- *notificações de evento* desacoplam produtores de evento dos consumidores de evento.

O segundo grande tema da CDI é a *tipificação forte*. As informações sobre as dependências, interceptores e decoradores de um 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 em código CDI, não é devido ao framework estar escondendo eles de você usando regras espertas de padronização—a chamada \"configuração por convenção\"—mas porque simplesmente não precisamos 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.

CDI 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.

Os *estereótipos* CDI 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 escopo, vínculações de interceptadores, qualificadores, etc, em um único pacote reutilizável. (Certamente, existe também o benefício de aconchegar algumas destas anotações).

Nós agora estamos prontos para verificar mais algumas funcionalidades avançadas de CDI. 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.

Métodos produtores

Métodos produtores permitem superarmos certas limitações que surgem quando um contêiner, em vez da aplicação, é responsável por instanciar objetos. Eles são também a forma mais fácil de integrar os objetos que não são beans ao ambiente CDI.

De acordo com a especificação:

> Um método produtor funciona como uma fonte de objetos a serem injetados, onde:
>
> - os objetos a serem injetados não necessitam ser instâncias de beans,
> - o tipo concreto dos objetos a serem injetados pode variar em tempo de execução ou
> - os objetos requerem alguma inicialização personalizada que não é realizada pelo construtor do bean

Por exemplo, métodos produtores permitem:

- expõe uma entidade JPA como um bean,
- expõe qualquer classe do JDK como um bean,
- definir vários beans, com diferentes escopos ou inicialização, para a mesma classe de implementação, ou
- varia a implementação de um tipo de bean em tempo de execução.

Em particular, métodos produtores permitem-nos utilizar polimorfismo em tempo de execução com CDI. Como vimos, os beans alternativos são uma solução para o problema do polimorfismo em tempo de implantação. Mas, uma vez que o sistema está implantado, a implementação do CDI é imutável. Um método produtor não tem essa limitação:

```
@SessionScoped
public class Preferences implements Serializable {
   private PaymentStrategyType paymentStrategy;
   ...
   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
       switch (paymentStrategy) {
           case CREDIT_CARD: return new CreditCardPaymentStrategy();
           case CHECK: return new CheckPaymentStrategy();
           case PAYPAL: return new PayPalPaymentStrategy();
           default: return null;
       }
   }
}
```

Considere o ponto de injeção:

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

Este ponto de injeção tem o mesmo tipo e anotações de qualificadoras que o método produtor. Assim, ele resolve para o método produtor utilizando as regras de injeção do CDI. O método produtor será invocado pelo contêiner para obter uma instância para servir esse ponto de injeção.

8.1. Escopo de um método produtor

O escopo padrão dos métodos produtores é `@Dependent`, e, por isso, serão chamados *toda vez* que o contêiner 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 de sessão. O método produtor não será invocado novamente na mesma sessão.

Nota

Um método produtor *não* herda o escopo do bean que declara o método. Existem dois beans diferentes aqui: o método produtor, e o bean que o declara. O escopo do método produtor determina como o método será invocado, e o ciclo de vida dos objetos retornados pelo método. O ecopo do bean que declara o método produtor determina o ciclo de vida do objeto no qual o método produtor é invocado.

8.2. Injeção em métodos produtores

Existe um problema potencial com o código acima. As implementações de `CreditCardPaymentStrategy` são instanciadas utilizando o operador `new` de Java. 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 bean:

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

Espere, o que ocorre se `CreditCardPaymentStrategy` for um bean com escopo de requisição? Assim o método produtor tem o efeito de "promover" a instância atual com escopo de requisição para escopo de sessão. Isso certamente é um erro! O objeto com escopo de requisição será destruído pelo contêiner antes de terminar a

sessão, mas a referência ao objeto será deixada "presa" ao escopo de sessão. Esse erro *não* será detectado pelo contêiner. Por isso, tome cuidado adicional ao retornar instâncias de bean em métodos produtores!

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

Mas uma solução mais comum é utilizar a anotação qualificadora especial `@New`.

8.3. Uso do `@New` em métodos produtores

Considere o seguinte método produtor:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New CheckPaymentStrategy 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, ao término da sessão.

8.4. Métodos destruidores

Alguns métodos produtores retornam objetos que requerem destruição explícita. Por exemplo, alguém precisa fechar esta conexão JDBC:

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

A destruição pode ser realizada por um *método destruidor* apropriado, definido pela mesma classe do método produtor:

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

O método destruidor deve ter ao menos um parâmetro, anotado com `@Disposes`, com o mesmo tipo e qualificadores do método produtor. O método destruidor é chamado automaticamente quando o contexto termina (neste caso, ao final da requisição), e este parâmetro recebe o objeto produzido pelo método produtor. Se o método produtor possui parâmetros adicionais, o contêiner procurará por um bean que satisfaça o tipo e qualificadores de cada parâmetro e o passará para o método automaticamente.

Interceptadores

A funcionalidade interceptador é definida na especificação Java Interceptors. CDI melhora esta funcionalidade com uma abordagem mais sofisticada, semântica e baseada em anotações para associar interceptores aos beans.

A especificação Interceptors define dois tipos de pontos de interceptação:

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

Adicionalmente, a especificação EJB define a interceptação do estouro de tempo.

Um *interceptador de método de negócio* se aplica a invocações de métodos do bean por clientes do bean:

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

Um *interceptador de chamadas de ciclo de vida* se aplica a invocações após algum evento do ciclo de vida do bean pelo contêiner:

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

Uma classe de interceptador pode interceptar tanto o ciclo de vida quanto métodos de negócio.

Um *interceptador de estouro de tempo* se aplica a invocações de métodos EJB limitadores de tempo pelo contêiner:

```
public class TimeoutInterceptor {
   @AroundTimeout
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

9.1. Bindings de interceptadores

Suponha que queremos declarar que algum de nossos beans são transacionais. A primeira coisa de que precisamos é um *tipo vinculador de interceptador* para especificar exatamente em quais beans estamos interessados:

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

Agora podemos especificar facilmente que nosso `ShoppingCart` é um objeto transacional:

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

Ou, se preferirmos, podemos especificar que apenas um método é transacional:

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

9.2. Implementando interceptadores (interceptors)

Isto é ótimo, mas em algum momento teremos que implementar o interceptador que fornece este aspecto de gerenciamento transacional. Tudo que precisamos fazer é criar um interceptador normal, e anotá-lo com `@Interceptor` e `@Transactional`.

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

Interceptadores podem tirar vantagem da injeção de dependência:

```
@Transactional @Interceptor
public class TransactionInterceptor {

    @Resource UserTransaction transaction;

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

}
```

Vários interceptadores podem usar o mesmo tipo vinculador.

9.3. Habiliatando interceptadores (interceptors)

Por padrão, todos os interceptadores estão desabilitados. Nós precisamos *habilitar* nosso interceptador no descritor `beans.xml` de um arquivo de beans. Esta ativação somente se aplica aos beans neste arquivo.

```
<beans
   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/beans_1_0.xsd">
   <interceptors>
      <class
>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans
>
```

Espere! Porque esse monte de elementos?

Bem, a declaração XML é atualmente uma *boa coisa*. E resolve dois problemas:

- o que nos possibilita especificar totalmente a ordem para todos os interceptores em nosso sistema, garantindo um comportamento determinístico, e
- o que nos permite habilitar ou desabilitar classes de interceptador em tempo de implantação.

Por exemplo, poderíamos especificar que nosso interceptador de segurança rode antes de nosso interceptador transacional.

```
<beans
   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/beans_1_0.xsd">
   <interceptors>
      <class
>org.mycompany.myapp.SecurityInterceptor</class>
      <class
>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans
>
```

Ou poderíamos desabilitá-los em nosso ambiente de teste apenas não os mencionando em `beans.xml`! Ah, tão simples.

9.4. Vinculando interceptadores com membros

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

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

CDI utilizará o valor de `requiresNew` para escolher entre fois diferentes interceptadores, `TransactionInterceptor` e `RequiresNewTransactionInterceptor`.

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

Agora podemos usar `RequiresNewTransactionInterceptor` assim:

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

Mas e se temos somente um interceptador e queremos que o contêiner ignore o valor de `requiresNew` ao vincular interceptadores? Talvez esta informação seja útil somente para a implementação do interceptador. Nós podemos usar a anotação `@Nonbinding`:

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

9.5. Múltiplas anotações de vinculação de interceptadores

Usualmente usamos combinações de tipos vinculadores de interceptadores para ligar múltiplos interceptadores a um bean. Por exemplo, a seguinte declaração poderia seria usada para vincular `TransactionInterceptor` e `SecurityInterceptor` ao mesmo bean:

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

No entanto, em casos muito complexos, um interceptador pode em si mesmo especificar algumas combinações de tipos vinculadores de interceptadores:

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

Então este interceptador poderia ser amarrado ao método `checkout()` usando qualquer uma das seguintes combinações:

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

```
}
```

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

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

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

9.6. Herança de tipos vinculadores de interceptadores

Uma limitação do suporte a anotações na linguagem Java é a falta de herança em anotações. Realmente, anotações poderiam ter esse reuso embutido, permitindo este tipo de coisa funcionasse:

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

Bem, felizmente, CDI contorna a falta desta funcionalidade em Java. Podemos anotar um tipo vinculador com outros tipos vinculadores de interceptadores (denominado como uma *meta-anotação*). As vinculações de interceptadores são transitivos — qualquer bean com o primeiro vínculo a um interceptador herda os vínculos a interceptadores declarados como meta-anotações.

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

Agora, qualquer bean anotado com `@Action` será amarrado tanto a `TransactionInterceptor` quanto a `SecurityInterceptor`. (E mesmo a `TransactionalSecureInterceptor`, se estiver presente.)

9.7. Uso de `@Interceptors`

A anotação `@Interceptors` definida pela especificação de interceptadores (e utilizada pelas especificações de managed bean e EJB) é ainda suportada em CDI.

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

No entanto, esta abordagem sofre os seguintes inconvenientes:

- a implementação do interceptador é duramente codificado no código de negócio,
- os interceptadores podem não ser facilmente disabilitados em tempo de implantação, e
- a ordem dos interceptadores não é global—ela é determinada pela ordem em que os interceptadores são listados a nível de classe.

Portanto, recomendamos o uso de vinculadores de interceptadores no estilo da CDI.

Decoradores

Interceptadores são um meio poderoso para capturar e separar preocupações *ortogonais* para a aplicação (e sistema de tipos). Qualquer interceptador é capaz de interceptar invocações de qualquer tipo Java. Isso os torna ideais para resolver questões técnicas, tais como gerenciamento de transação, segurança e registro de chamadas. 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 para *decoradores*. Um decorador intercepta invocações apenas para uma determinada interface Java e, portanto, é ciente de toda a semântica que acompanha esta interface. Visto que decoradores implementam diretamente operações com regras de negócios, isto torna eles uma ferramenta perfeita para modelar alguns tipos de questões de negócios. Isto também significa que um decorador não tem a generalidade de um interceptador. Decoradores não são capazes de resolver questões técnicas que atravessam muitos tipos diferentes. Interceptadores e decoradores, ambora similares em muitos aspectos, são complementares. Vamos ver alguns casos onde decoradores são bem convenientes.

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 beans diferentes em nosso sistema implementam a interface `Account`. No entanto, temos um requisito legal que, para qualquer tipo de conta, as grandes transações devem ser registadas pelo sistema em um registro (log) específico. Esse é um trabalho perfeito para um decorador.

Um decorador é um bean (possivelmente, até mesmo uma classe abstrata) que implementa o tipo que ele decora e é anotado com `@Decorator`.

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

O decorador implementa os métodos do tipo decorado que ele deseja interceptar.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      ...
   }
```

```
   public void deposit(BigDecimal amount);
      ...
   }
}
```

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

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

10.1. Objeto delegado

Decoradores possuem um ponto de injeção especial, chamado de *ponto de injeção delegado*, com o mesmo tipo dos beans que eles decoram e a anotação `@Delegate`. Deve haver exatamente um ponto de injeção delegado, que pode ser um parâmetro de construtor, um parâmetro de método inicializador ou um campo injetado.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;
   ...
}
```

Um decorador é vinculado a qualquer bean que:

- tenha o tipo do ponto de injeção delegado como um tipo de bean, e
- tenha todos os qualificadores que estão declarados no ponto de injeção delegado.

Este ponto de injeção delegado especifica que o decorador está vinculado a todos os beans que implementam `Account`:

```
@Inject @Delegate @Any Account account;
```

Um ponto de injeção delegado pode especificar qualquer número de anotações de qualificador. O decorador só será vinculado a beans com os mesmos qualificadores.

```
@Inject @Delegate @Foreign Account account;
```

O decorador pode invocar o objeto delegado, o que praticamente equivale a chamar `InvocationContext.proceed()` a partir de um interceptador. A principal diferença é que o decorador pode invocar *qualquer* método de negócio sobre o objeto delegado.

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

```
   @Inject @Delegate @Any 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) );
      }
   }
}
```

10.2. Habilitando decoradores

Por padrão, todos decoradores estão desabilitados. Nós precisamos *habilitar* nosso decorador no descritor `beans.xml` de um arquivo de beans. Esta ativação somente se aplica aos beans neste arquivo.

```
<beans
   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/beans_1_0.xsd">
   <decorators>
         <class
>org.mycompany.myapp.LargeTransactionDecorator</class>
   </decorators>
</beans
>
```

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

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

Eventos

Injeção de dependência possibilita baixo acoplamento permitindo que a implementação do bean type injetado alterne, seja durante a implantação ou em tempo de execução. Eventos vão mais a frente, permitem que beans interajam absolutamente com nenhuma dependência em tempo de compilação. Os *produtores* disparam eventos que são entregues a *observadores* de evento pelo contêiner.

Este esquema básico pode soar como o conhecido padrão observer/observable, mas existem algumas 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 notificações de evento que irão receber, e
- os observadores podem ser notificados imediatamente, ou podem especificar que a entrega do evento deve esperar o fim da transação corrente.

A funcionalidade de notificação de evento do CDI utiliza mais ou menos a mesma abordagem typesafe que já vimos no serviço de injeção de dependência.

11.1. Conteúdo dos eventos

O objeto de evento carrega estado do produtor para o consumidor. O objeto do evento é nada mais que uma instância de uma classe Java concreta. (A única restrição é que as classes de evento não podem conter variáveis de tipo). Um evento pode ter qualificadores atribuídos, permitindo que observadores o diferencie de outros eventos do mesmo tipo. Os qualificadores funcionam como seletores de tópico, possibilitando que um observador reduza o conjunto de eventos a observar.

Um qualificador de evento é apenas um qualificador normal, definido com `@Qualifier`. Aqui vai um exemplo:

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

11.2. Observadores de eventos

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

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

O parâmetro anotado é chamado de *parâmetro de evento*. O tipo do parâmetro de evento é a *classe de evento* observada, neste caso `Document`. O parâmetro de evento também pode especificar qualificadores.

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

Um método observador não precisa especificar qualificadores de evento—neste caso ele estará interessado em *todos* eventos de um tipo específico. Se ele especificar qualificadores, somente estará interessado em eventos que possuem estes qualificadores.

O método observador pode ter parâmetros adicionais, os quais são pontos de injeção:

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

11.3. Produtores de Eventos

Os produtores disparam eventos utilizando uma instância da interface `Event` parametrizada. Uma instância desta interface é obtida por injeção:

```
@Inject @Any Event<Document
> documentEvent;
```

Um produtor lança eventos chamando o método `fire()` da interface `Event`, passando o objeto de evento:

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

Este evento específico será entregue a todo método observador que:

- tenha um parâmetro de evento em que o objeto de evento (o `Document`) é especificado, e
- não especifique qualquer qualificador.

O contêiner simplesmente chama todos os métodos observadores, passando o objeto de evento como valor do parâmetro de evento. Se algum método observador lançar uma exceção, o contêiner para de chamar os métodos observadores, e a exceção é relançada pelo método `fire()`.

Os qualificadores podem ser aplicados a um evento em uma das seguintes formas:

- anotando o ponto de injeção `Event`, ou
- passando qualificadores para o método `select()` de `Event`.

Especificar os qualificadores no ponto de injeção é muito mais simples:

```
@Inject @Updated Event<Document
> documentUpdatedEvent;
```

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

- tenha um parâmetro evento em que o evento objeto é atribuído, e

- não possua qualquer qualificador de evento *exceto* para os qualificadores de evento que coincidam com os especificados no ponto de injeção de `Event`.

A desvantagem de anotar o ponto de injeção é que não podemos especificar o qualificador dinamicamente. CDI nos deixa obter uma instância de qualificador ao estender a classe auxiliadora `AnnotationLiteral`. Deste modo podemos passar o qualificador para o método `select()` de `Event`.

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

Eventos podem ter vários qualificadores de evento, agregados usando qualquer combinação de anotações no ponto de injeção de `Event` e passando instâncias qualificadoras ao método `select()`.

11.4. Métodos observadores condicionais

Por padrão, se nenhuma instância de um observador existir no contexto atual, o contêiner instanciará o observador para entregar um evento a ele. Este comportamento não é sempre desejável. Podemos querer entregar eventos somente para instâncias do observador que já existam nos contextos atuais.

Um observador condicional é especificado ao adicionar `receive = IF_EXISTS` na anotação `@Observes`.

```
public void refreshOnDocumentUpdate(@Observes(receive = IF_EXISTS) @Updated Document d) { ... }
```

Um bean com escopo `@Dependent` não pode ser um observador condicional, uma vez que ele nunca deveria ser chamado!

11.5. Qualificadores de eventos com membros

Um tipo qualificador de evento pode possuir anotações com membros:

```
@Qualifier
@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) { ... }
```

Os membros do tipo qualificador de evento podem ser especificados estaticamente pelo produtor de evento, por meio de anotações no ponto de injeção do notificador de evento:

```
@Inject @Role(ADMIN) Event<LoggedIn
```

```
> loggedInEvent;
```

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

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

O produtor de evento passa uma instância desta classe para `select()`:

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

11.6. Múltiplos qualificadores de eventos

Os tipos qualificadores de evento podem ser combinados, por exemplo:

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

Os observadores devem satisfazer completamente o tipo do qualificador final do evento. Assuma os seguintes observadores neste exemplo:

```
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) { ... }}}
```

O único observador a ser notificado será:

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

No entanto, se houver também um observador:

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

Este também deveria ser notificado, uma vez que `@Any` atua como um sinônimo para qualquer de todos qualificadores.

11.7. 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(@Observes(during = AFTER_SUCCESS) CategoryUpdateEvent event) { ... }
```

Existem cinco tipos de observadores transacionais:

- observadores `IN_PROGESS` são chamados imediatamente (padrão)

  observadores `AFTER_SUCCESS` são chamados durante a fase posterior à conclusão da transação, mas somente se a transação for concluída com sucesso

- observadores `AFTER_FAILURE` são chamados durante a fase posterior à conclusão da transação, mas somente se a transação não for concluída com sucesso
- observadores `AFTER_COMPLETION` são chamados durante a fase posterior à conclusão da transação
- observadores `BEFORE_COMPLETION` são chamados durante a fase anterior à conclusão da transação

Os observadores transacionais são muito importantes em um modelo de objetos stateful, porque o estado é muitas vezes mantido por mais de uma única transação atômica.

Imagine que fizemos cache do conjunto de resultados de uma consulta JPA no escopo de aplicação:

```
@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, precisamos atualizar o catálogo de `Product`. Mas devemos esperar até *depois* da transação ser concluída com sucesso antes de realizar essa atualização!

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

```
@Stateless
public class ProductManager {
   @PersistenceContext EntityManager em;
   @Inject @Any Event<Product
> productEvent;

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

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

E agora `Catalog` pode observar os eventos após a conclusão com sucesso da transação:

```
@ApplicationScoped @Singleton
public class Catalog {
   ...
   void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
      products.add(product);
   }

   void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
      products.remove(product);
   }
}
```

Estereótipos

A especificação CDI define um estereótipo da seguinte forma:

> Em muitos sistemas, a utilização de padrões arquiteturais produz um conjunto de papéis recorrentes de beans. Um estereótipo permite a um desenvolvedor de framework identificar esse papel e declarar alguns metadados comuns para beans com esse papel em um local centralizado.
>
> Um estereótipo encapsula qualquer combinação de:
>
> - um escopo padrão, e
> - um conjunto de bindings de interceptadores.
>
> Um estereótipo pode também especificar que:
>
> - todos os beans com o estereótipo possuem um nome padrão em EL, ou que
> - todos os beans com o estereótipo são alternativos.
>
> Um bean pode declarar nenhum, um ou vários estereótipos. As anotações de estereótipo podem ser aplicadas a uma classe de bean ou a um método ou campo produtor.

Um estereótipo é uma anotação anotada com `@Stereotype` que embrulha muitas outras anotações. Por exemplo, o seguinte estereótipo identifica classes de ação em algum framework MVC:

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

Nós utilizamos o estereótipo ao aplicar a anotação a um bean.

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

Claro que precisamos aplicar algumas outras anotações ao nosso estereótipo, ou então, não adicionaríamos muito valor a ele.

12.1. Escopo padrão para um estereótipo

Um estereótipo pode especificar um escopo padrão para todos os beans anotados com o estereótipo. Por exemplo:

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

Uma determinada ação pode ainda substituir este padrão se necessário:

```
@Dependent @Action
public class DependentScopedLoginAction { ... }
```

Naturalmente que substituir um único padrão não é muito útil. Mas lembre-se, estereótipos podem definir mais do que apenas o escopo padrão.

12.2. Bindings de interceptadores para estereótipos

Um estereótipo pode especificar um conjunto de vínculos a interceptadores a serem herdados por todos os beans com esse estereótipo.

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

Isso nos ajuda a manter os detalhes técnicos, como transações e segurança, ainda mais longe do código de negócio!

12.3. Padronização de nomes com estereótipos

Podemos especificar que todos os beans com um certo estereótipo tenham um nome EL padronizado quando um nome não é explicitamente definido por este bean. Tudo o que precisamos fazer é adicionar uma anotação `@Named` vazia:

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

Agora o bean `LoginAction` terá o nome `loginAction` por padrão.

12.4. Estereótipos alternativos

Um estereótipo pode indicar que todos os beans em que está aplicado são `@Alternatives`. Um *estereótipo alternativo* nos permite classificar beans por cenário de implantação.

```
@Alternative
@Stereotype
@Retention(RUNTIME)
```

```
@Target(TYPE)
public @interface Mock {}
```

Podemos aplicar um estereótipo alternativo para um conjunto inteiro de beans e ativá-los com apenas uma linha de código em `beans.xml`.

```
@Mock
public class MockLoginAction extends LoginAction { ... }
```

12.5. Empilhando estereótipos

Isto pode fundir sua mente um pouco, mas estereótipos podem declarar outros estereótipos, o que chamaremos de *empilhamento de estereótipos*. Você pode querer fazer isto se tiver dois estereótipos distintos que possuem seus próprios significados, mas em outra situação podem ter outro significado quando combinados.

Aqui vai um exemplo que combina os estereótipos `@Action` e `@Auditable`:

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

12.6. Estereótipos predefinidos

Já conhecemos dois estereótipos padrão definidos pela especificação CDI: `@Interceptor` e `@Decorator`.

CDI define um estereótipo adicional, `@Model`, que deverá ser usado frequentemente em aplicações web:

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

Em vez de utilizar managed beans JSF, basta anotar um bean com `@Model`, e utilizá-lo diretamente em sua página JSF!

Especialização, herança e alternativos

Quando você começa a desenvolver com CDI provavelmente estará lidando com apenas uma única implementação para cada bean type. Neste caso, é fácil entender como beans são selecionados para injeção. Conforme a complexidade de sua aplicação aumenta, várias ocorrências do mesmo bean type começam a aparecer, seja por existir várias implementações ou dois beans compartilharem uma mesma hierarquia (Java). É neste momento que você tem que começar a estudar as regras de especialização, herança e alternativos para lidar com dependências não satisfeitas ou ambíguas, ou para evitar que certos beans sejam chamados.

A especificação CDI reconhece dois cenários distintos e que um bean estende um outro:

- O bean suplementar *especializa* o bean inicial em certos cenários de implantação. Nestas implantações, o bean suplementar substitui completamente o primeiro, realizando o mesmo papel no sistema.
- O bean suplementar está simplesmente reutilizando a implementação Java, e de outro modo suporta nenhuma relação com o priemeiro bean. O bean inicial pode nem mesmo ter sido projetado para uso como um objeto contextual.

O segundo caso é o padrão assumido pela CDI. É possível ter dois beans no sistema com o mesmo bean type (interface ou classe pai). Conforme você aprendeu, você seleciona uma entre duas implementações usando qualificadores.

O primeiro caso é a exceção, e também requer mais cuidado. Em qualquer implantação, somente um bean pode realizar um dado papel em um momento. Isto significa que um bean precisa ser habilitado e outro desabilitado. Existem dois modificadores envolvidos: `@Alternative` e `@Specializes`. Vamos começar observando os alternativos e depois mostraremos as garantias que a especialização adiciona.

13.1. Utilizando estereótipos alternativos

CDI permite que você *sobrescreva* a implementação de um bean type durante a implantação utilizando um alternativo. Por exemplo, o seguinte bean fornece uma implementação padrão para a interface `PaymentProcessor`:

```
public class DefaultPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Mas em nosso ambiente simulado, não queremos efetivamente enviar ordens de pagamento para o sistema externo, dessa forma sobrescrevemos esta implementação de `PaymentProcessor` com um bean diferente:

```
public @Alternative
class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
```

```
}
```

or

```
public @Alternative
class StagingPaymentProcessor
      extends DefaultPaymentProcessor {
   ...
}
```

Já vimos como podemos habilitar este alternativo registrando sua classe no descritor `beans.xml`.

Mas suponha que temos muitos alternativos no ambiente simulado. Deveria ser mais conveniente habilitar todos eles de uma vez. então vamos tornar `@Staging` um estereótipo `@Alternative` e anotar os beans simuladores com este estereótipo. Você verá como este nível de indireção compensa. Primeiro, criamos o estereótipo:

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

Então substituímos a anotação `@Alternative` em nosso bean por `@Staging`:

```
@Staging
public class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Finalmente, ativamos o estereótipo `@Staging` no descritor `beans.xml`:

```
<beans
   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/beans_1_0.xsd">
   <alternatives>
         <stereotype
>org.mycompany.myapp.Staging</stereotype>
   </alternatives>
</beans
>
```

Agora, não importa quantos beans simuladores temos, eles serão habilitados todos de uma vez.

13.2. Um pequeno problema com alternativos

Quando habilitamos um alternativo, significa que a implementação padrão é desabilitada? Bem, não exatamente. Se a implementação padrão possui um qualificador, por exemplo `@LargeTransaction`, e o alternativo não possui, você pode ainda injetar a implementação padrão.

```
@Inject @LargeTransaction PaymentProcessor paymentProcessor;
```

Então não teremos substituído completamente a implementação padrão nesta implantação do sistema. O único modo que um bean pode substituir completamente um outro bean em todos os pontos de injeção é se ele implementa todos os beans types e declara todos os qualificadores deste outro bean. No entanto, se o outro bean declara um método produtor ou um método observador, então mesmo isto não é suficiente para assegurar que o outro bean nunca seja chamado! Precisamos de algo a mais.

CDI fornece uma funcionalidade especial, chamada *especialização*, que ajuda o desenvolvedor a evitar essas armadilhas. Especialização é um meio de informar o sistema de sua intenção de substituir completamente e desabilitar uma implementação de um bean.

13.3. Utilizando a especialização

Quando o objetivo é substituir uma implementação por uma outra, para ajudar a prevenir erros do desenvolvedor, o primeiro bean pode:

- estender diretamente a classe do outro bean, ou
- substituir diretamente o método produtor, no caso em que o outro bean for um método produtor, e então

declarar explicitamente que ele *especializa* o outro bean.

```
@Alternative @Specializes
public class MockCreditCardPaymentProcessor
      extends CreditCardPaymentProcessor {
   ...
}
```

Quando um bean está ativado e especializa um outro bean, este nunca é instanciado ou chamado pelo contêiner. Mesmo se o outro bean define um método produtor ou observador, o método nunca será chamado.

Então, como essa especialização funciona e o que tem a ver com herança?

Uma vez que estamos informando ao contêiner que nosso bean alternativo está destinado a ficar como um substituto da implementação padrão, a implementação alternativa automaticamente herda todos os qualificadores da implementação padrão. Desta maneira, em nosso exemplo, `MockCreditCardPaymentProcessor` herda os qualificadores `@Default` e `@CreditCard`.

Adicionalmente, se a implementação padrão declara um nome EL para o bean usando `@Named`, o nome é herdado pelo bean alternativo especializado.

Recursos do ambiente de componentes Java EE

Java EE 5 já tinha introduzido um suporte limitado a injeção de dependências, na forma de injeção de componentes do ambiente. Um recurso do ambiente de componentes é um componente Java EE, por exemplo um datasource JDBC, uma fila ou um tópico JMS, um contexto de persistência JPA, um EJB remoto ou um web service.

Naturalmente, agora existe uma leve incompatibilidade com o novo estilo de injeção de dependência em CDI. Mais notadamente, a injeção de componentes no ambiente se baseia em nomes para qualificar tipos ambíguos, e não há real consistência quanto à natureza dos nomes (algumas vezes um nome JNDI, outras vezes um nome de unidade de persistência, às vezes um link EJB, e por vezes um "nome mapeado" não-portável). Os campos produtores acabou se tornando um adaptador elegante para reduzir toda esta complexidade a um modelo comum e obter recursos do ambiente de componentes para participarem do sistema CDI como qualquer outra categoria de bean.

Os campos possuem uma dualidade em que eles podem tanto ser o alvo de uma injeção de componente do ambiente Java EE quanto ser declarado como um campo produtor da CDI. Por esse motivo, eles podem definir um mapeamento a partir de nomes textuais no ambiente de componentes, até uma combinação de tipo e qualificadores usados no mundo da injeção typesafe. Nós chamamos um campo produtor que representa uma referência a um objeto no ambiente de componentes Java EE de *recurso*.

14.1. Definindo um recurso

A especificação CDI utiliza o termo *recurso* para referir, genericamente, a qualquer das seguintes categorias de objeto que podem estar disponíveis no ambiente de componentes Java EE:

- `Datasources` do JDBC, `Queues`, `Topics` e `ConnectionFactorys` do JMS, `Sessions` do JavaMail e outros recursos transacionais incluindo os conectores JCA,
- `EntityManagers` e `EntityManagerFactorys` do JPA,
- EJBs remotos, e
- web services

Declaramos um recurso ao anotar um campo produtor com uma anotação de injeção de componentes de ambiente: `@Resource`, `@EJB`, `@PersistenceContext`, `@PersistenceUnit` ou `@WebServiceRef`.

```
@Produces @WebServiceRef(lookup="java:app/service/Catalog")
Catalog catalog;
```

```
@Produces @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
@CustomerDatabase Datasource customerDatabase;
```

```
@Produces @PersistenceContext(unitName="CustomerDatabase")
@CustomerDatabase EntityManager customerDatabasePersistenceContext;
```

```
@Produces @PersistenceUnit(unitName="CustomerDatabase")
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;
```

```
@Produces @EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

O campo pode ser estático (mas não final).

Uma declaração de recurso contém duas peças de informação:

- o nome JNDI, link EJB, nome de unidade de persistência, ou outro metadado necessário para obter uma referência ao recurso a partir do ambiente de componentes, e
- o tipo e os qualificadores que usaremos para injetar a referência dentro de nossos beans.

Nota

Pode parecer estranho declarar recursos no código Java. Não podem ser coisas específicas da implantação? Certamente, e é por isso que faz sentido declarar seus recursos em uma classe anotada com `@Alternative`.

14.2. Injeção typesafe de recursos

Este recursos agora podem ser injetados na maneira habitual.

```
@Inject Catalog catalog;
```

```
@Inject @CustomerDatabase Datasource customerDatabase;
```

```
@Inject @CustomerDatabase EntityManager customerDatabaseEntityManager;
```

```
@Inject @CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Inject PaymentService paymentService;
```

O tipo do bean e os qualificadores do recurso são determinados pela declaração do campo produtor.

Pode parecer trabalhoso ter que escrever estas declarações de campos produtores extras, apenas para ganhar um nível adicional de indireção. Você poderia muito bem usar uma injeção de componente do ambiente diretamente,

certo? Mas lembre-se que você vai usar recursos como o `EntityManager` em vários beans diferentes. Não é mais agradável e mais typesafe escrever

```
@Inject @CustomerDatabase EntityManager
```

em vez de

```
@PersistenceContext(unitName="CustomerDatabase") EntityManager
```

em todos os lugares?

Parte IV. CDI e o ecossistema Java EE

O terceiro tema de CDI é *integração*. Já vimos como CDI ajuda a integrar EJB e JSF, permitindo que EJBs sejam associados diretamente a páginas JSF. Isso é só o começo. Os serviços CDI são integrados dentro do núcleo da plataforma Java EE. Até mesmo session beans EJB podem tirar vantagem da injeção de dependência, do barramento de eventos, e o gerenciamento do ciclo de vida contextual que CDI fornece.

CDI também é projetado para trabalhar em conjunto com tecnologias fora da plataforma provendo pontos de integração dentro da plataforma Java EE por meio de uma SPI. Esta SPI coloca CDI como o alicerce para um novo ecosistema de extensões *portáveis* e integração com frameworks e tecnologias existentes. Os serviços CDI serão hábeis a abranger uma diversa coleção de tecnologias, tal como mecanismos de business process management (BPM), frameworks web existentes e modelos de componentes de facto padrão. Certamente, a plataforma Java EE nunca será capaz de padronizar todas as tecnologias interessantes que estão sendo usadas no mundo de desenvolvimento de aplicações Java, mas CDI torna mais fácil utilizar as tecnologias que ainda não fazem parte da plataforma suavemente dentro do ambiente Java EE.

Estamos prestes a ver como obter todas as vantagens da plataforma Java EE em uma aplicação que utiliza CDI. Também iremos conhecer resumidamente um conjunto de SPIs que são fornecidas para suportar extensões portáveis à CDI. Você pode nem mesmo precisar usar estas SPIs diretamente, mas não pense nisto como garantia. Você provavelmente irá usá-las indiretamente, cada vez que você utilizar uma extensão de terceiros, como o Seam.

Integração com o Java EE

CDI está plenamente integrada ao ambiente Java EE. Os beans possuem 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 outros componentes da plataforma, tais como servlets e message-driven Beans, que não são bens por si só.

15.1. Beans embutidos

No ambiente Java EE, o contêiner fornece os seguintes beans embutidos, todos com o qualificador `@Default`:

- o `UserTransaction` JTA corrente,
- um `Principal` representando a identidade do requisitante corrente,
- a `ValidationFactory` padrão da *Bean Validation* [http://jcp.org/en/jsr/detail?id=303], e
- um `Validator` para a `ValidationFactory` padrão.

Nota

A especificação CDI não requer que os objetos de contexto de servlet `HttpServletRequest`, `HttpSession` e `ServletContext` sejam expostos como beans injetáveis. Se você realmente quer ser capaz de injetar estes objetos, é fácil criar uma extensão portável para expô-los como beans. No entanto, recomendamos que o acesso direto a estes objetos estejam limitados a servlets, servlet filters e servlet event listeners, onde podem ser obtidos da maneira usual, tal como definido pela especificação Java Servlets. O objeto `FacesContext` também não é injetável. Você pode obtê-lo chamando `FacesContext.getCurrentInstance()`.

Nota

Oh, você *realmente* quer injetar o `FacesContext`? Tudo bem então, tente este método produtor:

```
class FacesContextProducer {
   @Produces @RequestScoped FacesContext getFacesContext() {
      return FacesContext.getCurrentInstance();
   }
}
```

15.2. Injetando recursos Java EE em um bean

Todos os beans gerenciados poder tirar vantagem da injeção de componentes do ambiente Java EE usando `@Resource`, `@EJB`, `@PersistenceContext`, `@PeristenceUnit` e `@WebServiceRef`. Nós já vimos uma porção de exemplos disto, embora não tenhamos prestado muita atenção no momento:

```
@Transactional @Interceptor
```

```
public class TransactionInterceptor {
   @Resource UserTransaction transaction;

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

```
@SessionScoped
public class Login implements Serializable {
   @Inject Credentials credentials;
   @PersistenceContext EntityManager userDatabase;
    ...
}
```

As chamadas Java EE `@PostConstruct` e `@PreDestroy` também são suportadas em todos os beans controlados. O método anotado com `@PostConstruct` é invocado após *todas* injeções serem realizadas.

Certamente, aconselhamos que a injeção de componentes do ambiente seja usada para definir recursos CDI, e que injeção typesafe seja usada no código da aplicação.

15.3. Invocando um bean a partir de um Servlet

É fácil usar um bean a partir de uma servlet no Java EE 6. Simplesmente injete o bean usando um campo ou injeção por um método inicializador.

```
public class Login extends HttpServlet {
   @Inject Credentials credentials;
   @Inject Login login;

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

}
```

Uma vez que instâncias de servlets são compartilhadas através de todas threads entrantes, o proxy cliente cuida do encaminhamento das invocações dos métodos do servlet para as instâncias corretas de `Credentials` e `Login` para a requisição e sessão HTTP atuais.

15.4. Invocando um bean a partir de um message-driven bean

A injeção CDI se aplica a todos EJBs, mesmo quando eles não são beans gerenciados. Em especial, você pode usar injeção CDI em message-driven beans, os quais são por natureza objetos não contextuais.

Você ainda pode usar bindings de interceptação da CDI em message-driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
   @Inject Inventory inventory;
   @PersistenceContext EntityManager em;

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

Note que existe nenhuma sessão ou contexto de conversação disponível quando uma mensagem é entregue a um message-driven bean. Apenas benas `@RequestScoped` e `@ApplicationScoped` estão disponíveis.

Mas como que beans *enviam* mensagens JMS?

15.5. 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.

Você pode usar campos e métodos produtores para preparar todos estes recursos para injeção em um bean:

```
public class OrderResources {
   @Resource(name="jms/ConnectionFactory")
   private ConnectionFactory connectionFactory;

   @Resource(name="jms/OrderQueue")
   private Queue orderQueue;

   @Produces @OrderConnection
   public Connection createOrderConnection() throws JMSException {
    return connectionFactory.createConnection();
   }

   public void closeOrderConnection(@Disposes @OrderConnection Connection connection)
         throws JMSException {
      connection.close();
   }

   @Produces @OrderSession
   public Session createOrderSession(@OrderConnection Connection connection)
         throws JMSException {
```

```
      return connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
   }

   public void closeOrderSession(@Disposes @OrderSession Session session)
         throws JMSException {
      session.close();
   }

   @Produces @OrderMessageProducer
   public MessageProducer createOrderMessageProducer(@OrderSession Session session)
         throws JMSException {
      return session.createProducer(orderQueue);
   }

   public void closeOrderMessageProducer(@Disposes @OrderMessageProducer MessageProducer producer)
         throws JMSException {
      producer.close();
   }
}
```

Neste exemplo, podemos injetar apenas `MessageProducer`, `Connection` ou `QueueSession`:

```
@Inject Order order;
@Inject @OrderMessageProducer MessageProducer producer;
@Inject @OrderSession QueueSession orderSession;

public void sendMessage() {
   MapMessage msg = orderSession.createMapMessage();
   msg.setLong("orderId", order.getId());
   ...
   producer.send(msg);
}
```

O ciclo de vida dos objetos JMS injetados são completamente controlados pelo contêiner.

15.6. Empacotamento e implantação

CDI não define qualquer arquivo de implantação especial. Você pode empacotar beans em jars, ejb jars ou wars—qualquer local no classpath da aplicação em implantação. Entretanto, o arquivo deve ser um "arquivo de beans". Isto significa que arquivos contendo beans *deve* incluir um arquivo nomeado como `beans.xml` no diretório `META-INF` do classpath ou no diretório `WEB-INF` (para arquivos war). O arquivo pode ser vazio. Os beans implantados em arquivos que não possuam um `beans.xml` não estarão disponíveis para uso na aplicação.

Em um contêiner EJB embutido, os beans podem ser implantados em qualquer local em que EJBs podem ser implantados. Novamente, cada local deve conter um arquivo `beans.xml`.

Extensões portáveis

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

- 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
- novas tecnologias baseadas no modelo de programação da CDI.

Mais formalmente, de acordo com a especificação:

> Uma extensão portável pode integrar com o contêiner:
>
> - Fornecendo seus próprios beans, interceptadores e decoradores ao contêiner
> - Injetando dependências em seus próprios objetos usando o serviço de injeção de dependência
> - Fornecendo uma implementação de contexto para um escopo personalizado
> - Aumentando ou sobrescrevendo os metadados das anotações com metadados de algum outro lugar

16.1. Criando uma `Extension`

O primeiro passo na criação de uma extensão portável é escrever uma classe que implementa `Extension`. Esta interface indicadora não define qualquer método, mas é necessária para satisfazer os requisitos da arquitetura de provedores de serviço da Java SE.

```
class MyExtension implements Extension { ... }
```

Agora, precisamos registrar nossa extensão como um provedor de serviço criando um arquivo nomeado como `META-INF/services/javax.enterprise.inject.spi.Extension`, o qual contém o nome da classe de nossa extensão:

```
org.mydomain.extension.MyExtension
```

Uma extensão não é um bean, exatamente, já que é instanciada pela contêiner durante o processo de inicialização, antes de qualquer bean ou contexto existir. Entretanto, pode ser injetada em outros beans uma vez que o processo de inicialização estiver completo.

```
@Inject
MyBean(MyExtension myExtension) {
   myExtension.doSomething();
}
```

E, como beans, extensões podem ter métodos observadores. Geralmente, os métodos observadores observam *eventos do ciclo de vida do contêiner*.

16.2. Eventos do ciclo de vida do contêiner

Durante o processo de inicialização, o contêiner dispara uma série de eventos, incluindo:

- `BeforeBeanDiscovery`
- `ProcessAnnotatedType`
- `ProcessInjectionTarget` e `ProcessProducer`
- `ProcessBean` e `ProcessObserverMethod`
- `AfterBeanDiscovery`
- `AfterDeploymentValidation`

Extensões podem observar estes eventos:

```
class MyExtension implements Extension {

   void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
      Logger.global.debug("beginning the scanning process");
   }

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {
      Logger.global.debug("scanning type: " + pat.getAnnotatedType().getJavaClass().getName());
   }

   void afterBeanDiscovery(@Observes AfterBeanDiscovery abd) {
      Logger.global.debug("finished the scanning process");
   }

}
```

De fato, a extensão pode fazer muito mais que apenas observar. A extensão pode modificar o metamodelo do contêiner e mais. Aqui está um exemplo bem simples:

```
class MyExtension implements Extension {

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {
      //tell the container to ignore the type if it is annotated @Ignore
      if ( pat.getAnnotatedType().isAnnotionPresent(Ignore.class) ) pat.veto();
   }

}
```

O método observador pode injetar um `BeanManager`.

```
<T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat, BeanManager beanManager) { ... }
```

16.3. O objeto `BeanManager`

O nervo central para extender CDI é o objeto `BeanManager`. A interface `BeanManager` nos permite, programaticamente, obter beans, interceptadores, decoradores, observadores e contextos.

```
public interface BeanManager {
   public Object getReference(Bean<?> bean, Type beanType, CreationalContext<?> ctx);
   public Object getInjectableReference(InjectionPoint ij, CreationalContext<?> ctx);
   public <T
> CreationalContext<T
> createCreationalContext(Contextual<T
> contextual);
   public Set<Bean<?>
> getBeans(Type beanType, Annotation... qualifiers);
   public Set<Bean<?>
> getBeans(String name);
   public Bean<?> getPassivationCapableBean(String id);
   public <X
> Bean<? extends X
> resolve(Set<Bean<? extends X
>
> beans);
   public void validate(InjectionPoint injectionPoint);
   public void fireEvent(Object event, Annotation... qualifiers);
   public <T
> Set<ObserverMethod<? super T
>
> resolveObserverMethods(T event, Annotation... qualifiers);
   public List<Decorator<?>
> resolveDecorators(Set<Type
> types, Annotation... qualifiers);
   public List<Interceptor<?>
> resolveInterceptors(InterceptionType type, Annotation... interceptorBindings);
   public boolean isScope(Class<? extends Annotation
> annotationType);
   public boolean isNormalScope(Class<? extends Annotation
> annotationType);
   public boolean isPassivatingScope(Class<? extends Annotation
> annotationType);
   public boolean isQualifier(Class<? extends Annotation
> annotationType);
   public boolean isInterceptorBinding(Class<? extends Annotation
> annotationType);
   public boolean isStereotype(Class<? extends Annotation
> annotationType);
   public Set<Annotation
> getInterceptorBindingDefinition(Class<? extends Annotation
> bindingType);
```

```
   public Set<Annotation
> getStereotypeDefinition(Class<? extends Annotation
> stereotype);
   public Context getContext(Class<? extends Annotation
> scopeType);
   public ELResolver getELResolver();
   public ExpressionFactory wrapExpressionFactory(ExpressionFactory expressionFactory);
   public <T
> AnnotatedType<T
> createAnnotatedType(Class<T
> type);
   public <T
> InjectionTarget<T
> createInjectionTarget(AnnotatedType<T
> type);
}
```

Qualquer bean ou outro componente Java EE que suporte injeção pode obter uma instância do `BeanManager` via injeção:

```
@Inject BeanManager beanManager;
```

Os componentes Java EE podem obter uma instância de `BeanManager` a partir de JNDI procurando pelo nome `java:comp/BeanManager`. Qualquer operação de `BeanManager` pode ser chamada a qualquer momento durante a execução da aplicação.

Vamos estudar algumas das interfaces expostas pelo `BeanManager`.

16.4. A interface `InjectionTarget`

A primeira coisa que um desenvolvedor de framework vai procurar na extensão portável SPI é uma maneira de injetar beans CDI em objetos que não estão sob o controle de CDI. A interface `InjectionTarget` torna isto muito fácil.

Nota

Nós recomendamos que frameworks deixem CDI assumir o trabalho de efetivamente instanciar os objetos controladod pelo framework. Deste modo, os objetos controlados pelo framework podem tirar vantagem da injeção pelo construtor. No entanto, se o framework requer o uso de um construtor com uma assinatura especial, o próprio framework precisará instanciar o objeto, e então somente injeção por método e campo serão suportados.

```
//get the BeanManager from JNDI
BeanManager beanManager = (BeanManager) new InitialContext().lookup("java:comp/BeanManager");

//CDI uses an AnnotatedType object to read the annotations of a class
AnnotatedType<SomeFrameworkComponent
> type = beanManager.createAnnotatedType(SomeFrameworkComponent.class);

//The extension uses an InjectionTarget to delegate instantiation, dependency injection
```

```
//and lifecycle callbacks to the CDI container
InjectionTarget<SomeFrameworkComponent
> it = beanManager.createInjectionTarget(type);

//each instance needs its own CDI CreationalContext
CreationalContext ctx = beanManager.createCreationalContext(null);

//instantiate the framework component and inject its dependencies
SomeFrameworkComponent instance = it.produce(ctx);  //call the constructor
it.inject(instance, ctx);  //call initializer methods and perform field injection
it.postConstruct(instance);  //call the @PostConstruct method

...

//destroy the framework component instance and clean up dependent objects
it.preDestroy(instance);  //call the @PreDestroy method
it.dispose(instance);  //it is now safe to discard the instance
ctx.release();  //clean up dependent objects
```

16.5. A interface Bean

Instâncias da interface `Bean` representam beans. Existe uma instância de `Bean` registrada com o objeto `BeanManager` para todos os beans da aplicação. Há ainda objetos `Bean` representando interceptadores, decorados e métodos produtores.

The `Bean` interface exposes all the interesting things we discussed in *Seção 2.1, "A anatomia de um bean"*.

```
public interface Bean<T
> extends Contextual<T
> {
   public Set<Type
> getTypes();
   public Set<Annotation
> getQualifiers();
   public Class<? extends Annotation
> getScope();
   public String getName();
   public Set<Class<? extends Annotation
>
> getStereotypes();
   public Class<?> getBeanClass();
   public boolean isAlternative();
   public boolean isNullable();
   public Set<InjectionPoint
> getInjectionPoints();
}
```

Há uma maneira fácil de descobrir quais beans existem na aplicação:

```
Set<Bean<?>
> allBeans = beanManager.getBeans(Obect.class, new AnnotationLiteral<Any
>() {});
```

A interface `Bean` torna possível a uma extensão portável fornecer suporte a novos tipos de beans, além daqueles definidos pela especificação CDI. Por exemplo, poderíamos usar a interface `Bean` para permitir que os objetos gerenciados por um outro framework possam ser injetados nos beans.

16.6. Registrando um `Bean`

O tipo mais comum de extensão portável em CDI é para registro de beans no contêiner.

Neste exemplo, tornaremos uma classe do framework, `SecurityManager` disponível para injeção. Para tornar as coisas um pouco mais interessantes, vamos delegar de volta ao `InjectionTarget` do contêiner para realizar a instanciação e injeção das instâncias de `SecurityManager`.

```
public class SecurityManagerExtension implements Extension {

    void afterBeanDiscovery(@Observes AfterBeanDiscovery abd, BeanManager bm) {

        //use this to read annotations of the class
        AnnotatedType<SecurityManager
> at = bm.createAnnotatedType(SecurityManager.class);

        //use this to instantiate the class and inject dependencies
        final InjectionTarget<SecurityManager
> it = bm.createInjectionTarget(at);

        abd.addBean( new Bean<SecurityManager
>() {

            @Override
            public Class<?> getBeanClass() {
                return SecurityManager.class;
            }

            @Override
            public Set<InjectionPoint
> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public String getName() {
                return "securityManager";
            }

            @Override
            public Set<Annotation
> getQualifiers() {
                Set<Annotation
> qualifiers = new HashSet<Annotation
>();
                qualifiers.add( new AnnotationLiteral<Default
>() {} );
                qualifiers.add( new AnnotationLiteral<Any
>() {} );
                return qualifiers;
            }

            @Override
```

```
            public Class<? extends Annotation
> getScope() {
                return SessionScoped.class;
            }

            @Override
            public Set<Class<? extends Annotation
>
> getStereotypes() {
                return Collections.emptySet();
            }

            @Override
            public Set<Type
> getTypes() {
                Set<Type
> types = new HashSet<Type
>();
                types.add(SecurityManager.class);
                types.add(Object.class);
                return types;
            }

            @Override
            public boolean isAlternative() {
                return false;
            }

            @Override
            public boolean isNullable() {
                return false;
            }

            @Override
            public SecurityManager create(CreationalContext<SecurityManager
> ctx) {
                SecurityManager instance = it.produce(ctx);
                it.inject(instance, ctx);
                it.postConstruct(instance);
                return instance;
            }

            @Override
            public void destroy(SecurityManager instance,
                                CreationalContext<SecurityManager
> ctx) {
                it.preDestroy(instance);
                it.dispose(instance);
                ctx.release();
            }

        } );
    }

}
```

Mas uma extensão portável também pode se misturar com beans que são descobertos automaticamente pelo contêiner.

16.7. Envolvendo um `AnnotatedType`

Uma das coisas mais interessantes que uma classe de extensão pode fazer é processar as anotações de uma classe de bean *antes* do contêiner construir seu metamodelo.

Vamos começar com um exemplo de uma extensão que fornece suporte ao uso de `@Named` a nível de pacote. O nome em nível de pacote é utilizado para qualificar os nomes EL de todos os beans definidos neste pacote. A extensão portável utiliza o evento `ProcessAnnotatedType` para envolver o objeto `AnnotatedType` e sobrescrever o `value()` da anotação `@Named`.

```
public class QualifiedNameExtension implements Extension {

    <X
> void processAnnotatedType(@Observes ProcessAnnotatedType<X
> pat) {

        //wrap this to override the annotations of the class
        final AnnotatedType<X
> at = pat.getAnnotatedType();

        AnnotatedType<X
> wrapped = new AnnotatedType<X
>() {

            @Override
            public Set<AnnotatedConstructor<X
>
> getConstructors() {
                return at.getConstructors();
            }

            @Override
            public Set<AnnotatedField<? super X
>
> getFields() {
                return at.getFields();
            }

            @Override
            public Class<X
> getJavaClass() {
                return at.getJavaClass();
            }

            @Override
            public Set<AnnotatedMethod<? super X
>
> getMethods() {
                return at.getMethods();
            }

            @Override
            public <T extends Annotation
> T getAnnotation(final Class<T
```

```
> annType) {
                if ( Named.class.equals(annType) ) {
                    class NamedLiteral
                            extends AnnotationLiteral<Named
>
                            implements Named {
                        @Override
                        public String value() {
                            Package pkg = at.getClass().getPackage();
                            String unqualifiedName = at.getAnnotation(Named.class).value();
                            final String qualifiedName;
                            if ( pkg.isAnnotationPresent(Named.class) ) {
                                qualifiedName = pkg.getAnnotation(Named.class).value()
                                      + '.' + unqualifiedName;
                            }
                            else {
                                qualifiedName = unqualifiedName;
                            }
                            return qualifiedName;
                        }
                    }
                    return (T) new NamedLiteral();
                }
                else {
                    return at.getAnnotation(annType);
                }
            }

            @Override
            public Set<Annotation
> getAnnotations() {
                return at.getAnnotations();
            }

            @Override
            public Type getBaseType() {
                return at.getBaseType();
            }

            @Override
            public Set<Type
> getTypeClosure() {
                return at.getTypeClosure();
            }

            @Override
            public boolean isAnnotationPresent(Class<? extends Annotation
> annType) {
                return at.isAnnotationPresent(annType);
            }

        };

        pat.setAnnotatedType(wrapped);
    }

}
```

Aqui está um segundo exemplo, o qual adiciona a anotação `@Alternative` a qualquer classe que implementa uma certa interface `Service`.

```
class ServiceAlternativeExtension implements Extension {

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {

      final AnnotatedType<T
> type = pat.getAnnotatedType();

      if ( Service.class.isAssignableFrom( type.getJavaClass() ) ) {

         //if the class implements Service, make it an @Alternative
         AnnotatedType<T
> wrapped = new AnnotatedType<T
>() {

            @Override
            public boolean isAnnotationPresent(Class<? extends Annotation
> annotationType) {
               return annotationType.equals(Alternative.class) ?
                  true : type.isAnnotationPresent(annotationType);
            }

            //remaining methods of AnnotatedType
            ...
         }

         pat.setAnnotatedType(wrapped);
      }
   }

}
```

O `AnnotatedType` não é a única coisa que pode ser embrulhada por uma extensão.

16.8. Envolvendo um `InjectionTarget`

A interface `InjectionTarget` expõe operações para produzir e eliminar uma instância de um componente, injetando suas dependências e invocando suas callbacks do ciclo de vida. Uma extensão portável pode embrulhar `InjectionTarget` para qualquer componente Java EE que suporte injeção, permitindo que ela intercepte qualquer uma destas operações ao serem invocadas pelo contêiner.

Aqui está uma extensão CDI portável que lê valores de arquivos de propriedades e configura campos de componentes Java EE, incluindo servlets, EJBs, managed beans, interceptadores e mais outros. Neste exemplo, as propriedades de uma classe `org.mydomain.blog.Blogger` vão em um recurso nomeado como `org/mydomain/blog/Blogger.properties`, e o nome de uma propriedade deve casar com o nome do campo a ser configurado. Assim `Blogger.properties` deve conter:

```
firstName=Gavin
lastName=King
```

A extensão portável funciona ao envolver a `InjectionTarget` do contêiner e definindo os valores dos campos a partir do método `inject()`.

```
public class ConfigExtension implements Extension {

        <X
> void processInjectionTarget(@Observes ProcessInjectionTarget<X
> pit) {

                //wrap this to intercept the component lifecycle
            final InjectionTarget<X
> it = pit.getInjectionTarget();

        final Map<Field, Object
> configuredValues = new HashMap<Field, Object
>();

        //use this to read annotations of the class and its members
        AnnotatedType<X
> at = pit.getAnnotatedType();

        //read the properties file
        String propsFileName = at.getClass().getSimpleName() + ".properties";
        InputStream stream = at.getJavaClass().getResourceAsStream(propsFileName);
        if (stream!=null) {

            try {
                Properties props = new Properties();
                props.load(stream);
                for (Map.Entry<Object, Object
> property : props.entrySet()) {
                    String fieldName = property.getKey().toString();
                    Object value = property.getValue();
                    try {
                        Field field = at.getJavaClass().getField(fieldName);
                        field.setAccessible(true);
                        if ( field.getType().isAssignableFrom( value.getClass() ) ) {
                            configuredValues.put(field, value);
                        }
                        else {
                            //TODO: do type conversion automatically
                            pit.addDefinitionError( new InjectionException(
                                    "field is not of type String: " + field ) );
                        }
                    }
                    catch (NoSuchFieldException nsfe) {
                        pit.addDefinitionError(nsfe);
                    }
                    finally {
                        stream.close();
                    }
                }
            }
            catch (IOException ioe) {
                pit.addDefinitionError(ioe);
            }
        }
```

```
        InjectionTarget<X
> wrapped = new InjectionTarget<X
>() {

            @Override
            public void inject(X instance, CreationalContext<X
> ctx) {
                it.inject(instance, ctx);

                //set the values onto the new instance of the component
                for (Map.Entry<Field, Object
> configuredValue: configuredValues.entrySet()) {
                    try {
                        configuredValue.getKey().set(instance, configuredValue.getValue());
                    }
                    catch (Exception e) {
                        throw new InjectionException(e);
                    }
                }
            }

            @Override
            public void postConstruct(X instance) {
                it.postConstruct(instance);
            }

            @Override
            public void preDestroy(X instance) {
                it.dispose(instance);
            }

            @Override
            public void dispose(X instance) {
                it.dispose(instance);
            }

            @Override
            public Set<InjectionPoint
> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public X produce(CreationalContext<X
> ctx) {
                return it.produce(ctx);
            }

        };

        pit.setInjectionTarget(wrapped);

    }

}
```

Há muito mais sobre extensão portável SPI do que temos discutido aqui. Verifique a especificação CDI ou seu Javadoc para mais informações. Por agora, apenas mencionaremos mais um ponto de extensão.

16.9. A interface `Context`

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

```
public interface Context {
   public Class<? extends Annotation
> getScope();
   public <T
> T get(Contextual<T
> contextual, CreationalContext<T
> creationalContext);
   public <T
> T get(Contextual<T
> contextual);
   boolean isActive();
}
```

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

Próximos passos

Devido a CDI ser tão nova, não há ainda uma grande quantidade de informação disponível online. Isto mudará com o tempo. Independentemente, a especificação CDI permanece como a autoridade para informações sobre CDI. A especificação possui menos de 100 páginas e é bastante legível (não se preocupe, não são como o manual de seu aparelho Blue-ray). Certamente, ela cobre muitos detalhes que ignoramos aqui. A especificação está disponível em *JSR-299 page* [http://jcp.org/en/jsr/detail?id=299] no website do JCP.

A implementação de referência de CDI, Weld, está sendo desenvolvida no *projeto Seam* [http://seamframework.org/Weld]. O time de desenvolvimento da RI e o líder da especificação CDI blogam em *in.relation.to* [http://in.relation.to]. Este guia foi originalmente baseado em uma série de postagens publicadas neste blog enquanto a especificação estava sendo desenvolvida. É provavelmente a melhor fonte de informação sobre o futuro de CDI, Weld e Seam.

Nós encorajamos você a seguir a lista de discussão *weld-dev* [https://lists.jboss.org/mailman/listinfo/weld-dev] e se envolver no *desenvolvimento* [http://seamframework.org/Weld/Development]. Se você está lendo este guia, você certamente tem algo a oferecer.

Parte V. Guia de Referência do Weld

Weld é a implementação de referência da JSR-299 e é utilizada pelo JBoss AS e pelo Glassfish para prover serviços CDI às aplicações Java Enterprise Edition (Java EE). Weld também 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).

Você também pode querer dar uma olhada no projeto Weld Extensions que fornece extensões portáveis à CDI (tal como injeção de log e anotações adicionais para criação de beans), e Seam, que fornece integrações com outras camadas de visão (tal como GWT e Wicket) e outros frameworks (como Drools) bem como extensões ao ecossistema (como suporte a segurança).

If you want to get started quickly using Weld (and, in turn, CDI) with JBoss AS, GlassFish or Tomcat and experiment with one of the examples, take a look at *Capítulo 6, Iniciando com o Weld*. Otherwise read on for a exhaustive discussion of using Weld in all the environments and application servers it supports and the Weld extensions.

Servidores de aplicação e ambientes suportados pelo Weld

18.1. Utilizando Weld com o JBoss AS

Se você está usando JBoss AS 6.0, nenhuma configuração adicional é necessária para usar Weld (ou CDI para este caso). Tudo que você precisa fazer é tornar sua aplicação em um bean archive adicionando `META-INF/beans.xml` ao classpath ou `WEB-INF/beans.xml` à raiz web!

Nota

Adicionalmente, a Servlet do Weld suporta o JBoss EAP 5.1, usando a variante `jboss5` da Servlet do Weld.

18.2. GlassFish

O Weld também está embutido no GlassFish a partir da V3 e posteriores. Já que o GlassFish V3 é a implementação de referência da Java EE 6, ele deve suportar todas as funcionalidades da CDI. Qual a melhor maneira para que o GlassFish suporte estas funcionalidade do que usar Weld, a implementação de referência da JSR-299? Apenas empacote sua aplicação CDI e implante.

18.3. Servlet containers (como o Tomcat ou Jetty)

Enquanto a JSR-299 não requer suporte a ambientes servlet, o Weld pode ser utilizado em qualquer contêiner Servlet, como o Tomcat 6.0 ou Jetty 6.1.

Nota

Há uma limitação maior no uso de um contêiner servlet. O Weld não suporta a implantação de session beans, injeção usando `@EJB` ou `@PersistenceContext`, ou a utilização de eventos transacionais em contêineres servlet. Para funcionalidades corporativas como estas, você deveria realmente procurar um servidor de aplicações Java EE.

O Weld pode ser usado como uma biblioteca de aplicação web em um contêiner Servlet. Você deve colocar `weld-servlet.jar` dentro do diretório `WEB-INF/lib` relativo à raiz web. `weld-servlet.jar` é um "uber-jar", significando que ele empacota todas as partes do Weld e CDI necessárias para rodar em um contêiner servlet, para sua conveniência. Alternativamente, você pode usar seus jars componentes. Uma lista de dependências pode ser encontrada no arquivo `META-INF/DEPENDENCIES.txt` dentro do artefato `weld-servlet.jar`.

Você também precisa especificar explicitamente o servlet listener (usado para iniciar o Weld e controlar a interação com as requisições) no `web.xml`:

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

```
>
```

18.3.1. Tomcat

O Tomcat possui um JNDI somente leitura, assim o Weld não pode vincular automaticamente a extensão SPI do BeanManager. Para vincular o BeanManager ao JNDI, você deve preencher `META-INF/context.xml` na raiz web com o seguinte conteúdo:

```
<Context>
   <Resource name="BeanManager"
      auth="Container"
      type="javax.enterprise.inject.spi.BeanManager"
      factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context
>
```

e torná-lo disponível para a sua implantação, acrescentando isto no final do `web.xml`:

```
<resource-env-ref>
   <resource-env-ref-name
>BeanManager</resource-env-ref-name>
   <resource-env-ref-type>
      javax.enterprise.inject.spi.BeanManager
   </resource-env-ref-type>
</resource-env-ref
>
```

O Tomcat somente permite que você vincule entradas em `java:comp/env`, assim o BeanManager estará disponível em `java:comp/env/BeanManager`

Weld também suporta injeção de Servlet no Tomcat 6.

18.3.2. Jetty

Como o Tomcat, o Jetty possui um JNDI somente leitura, assim o Weld não pode vincular automaticamente o BeanManager. Para vincular o BeanManager ao JNDI no Jetty 6, você deve preencher `WEB-INF/jetty-env.xml` com o seguinte conteúdo:

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://jetty.mortbay.org/configure.dtd">
<Configure id="webAppCtx" class="org.mortbay.jetty.webapp.WebAppContext">
   <New id="BeanManager" class="org.mortbay.jetty.plus.naming.Resource">
      <Arg
><Ref id="webAppCtx"/></Arg
>
      <Arg
>BeanManager</Arg>
      <Arg>
         <New class="javax.naming.Reference">
            <Arg
```

```
>javax.enterprise.inject.spi.BeanManager</Arg
>
            <Arg
>org.jboss.weld.resources.ManagerObjectFactory</Arg>
            <Arg/>
          </New>
      </Arg>
   </New>
</Configure
>
```

O Jetty 7 foi movido para a Fundação Eclipse; se você está usando o Jetty 7 coloque o seguinte conteúdo em seu `WEB-INF/jetty-env.xml`:

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://www.eclipse.org/jetty/configure.dtd">

<Configure id="webAppCtx" class="org.eclipse.jetty.webapp.WebAppContext">
    <New id="BeanManager" class="org.eclipse.jetty.plus.jndi.Resource">
        <Arg
> <Ref id="webAppCtx"/> </Arg>
        <Arg
>BeanManager</Arg>
        <Arg>
            <New class="javax.naming.Reference">
                <Arg
>javax.enterprise.inject.spi.BeanManager</Arg>
                <Arg
>org.jboss.weld.resources.ManagerObjectFactory</Arg>
                <Arg/>
            </New>
        </Arg>
    </New>
</Configure
>
```

Assim como no Tomcat, você precisa torná-lo disponível em sua implantação, acrescentando isto ao final do `web.xml`:

```
<resource-env-ref>
   <resource-env-ref-name
>BeanManager</resource-env-ref-name>
   <resource-env-ref-type>
      javax.enterprise.inject.spi.BeanManager
   </resource-env-ref-type>
</resource-env-ref
>
```

Note que o Jetty não possui suporte já existente a um `javax.naming.spi.ObjectFactory` como o Tomcat, assim é necessário criar manualmente o `javax.naming.Reference` para envolvê-lo.

O Jetty somente permite que você vincule entradas em `java:comp/env`, assim o BeanManager estará disponível em `java:comp/env/BeanManager`

O Weld também suporta injeção em Servlet no Jetty 6. Para habilitar isto, adicione o arquivo `META-INF/jetty-web.xml` com o seguinte conteúdo em seu war:

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://jetty.mortbay.org/configure.dtd">
<Configure id="webAppCtx" class="org.mortbay.jetty.webapp.WebAppContext">
   <Call class="org.jboss.weld.environment.jetty.WeldServletHandler" name="process">
      <Arg
><Ref id="webAppCtx"/></Arg>
   </Call>
</Configure
>
```

18.4. Java SE

Em adição a uma integração melhorada da pilha Java Enterprise, a especificação "Contexts and Dependency Injection for the Java EE platform" também define um framework de injeção de dependência em estado da arte e typesafe, o qual pode se comprovar útil em uma ampla variedade de tipos de aplicação. Para ajudar desenvolvedores tirar vantagem disto, o Weld fornece um meio simples de ser executado no ambiente Java Standard Edition (SE) independentemente de qualquer API da Java EE.

Quando executando no ambiente SE as seguintes funcionalidades do Weld estão disponíveis:

- Managed beans com os callbacks `@PostConstruct` e `@PreDestroy` do ciclo de vida
- Injeção de dependência com qualificadores e alternativos
- Os escopos `@Application`, `@Dependent` e `@Singleton`
- Interceptadores e decoradores
- Estereótipos
- Eventos
- Suporte a extensão portável

Beans EJB não são suportados.

18.4.1. Módulo CDI SE

O Weld fornece uma extensão que inicializa um gerenciador de beans CDI em Java SE, registrando automaticamente todos os beans simples encontrados no classpath. Os parâmetros de linha de comando podem ser injetados usando uma das seguintes formas:

```
@Inject @Parameters List<String
> params;
```

```
@Inject @Parameters String[] paramsArray;
```

A segunda forma é útil para compatibilidade com classes existentes.

Nota

Os parâmetros de linha de comando não ficaram disponíveis para injeção até que o evento `ContainerInitialized` seja disparado. Se você precisa acessar os parâmetros durante a inicialização você pode fazer assim por meio do método `public static String[] getParameters()` em `StartMain`.

Aqui está um exemplo de uma simples aplicação CDI SE:

```
@Singleton
public class HelloWorld
{
   public void printHello(@Observes ContainerInitialized event, @Parameters List<String
> parameters) {
        System.out.println("Hello " + parameters.get(0));
   }
}
```

18.4.2. Inicializando aplicações CDI SE

Aplicações CDI SE podem ser inicializadas das seguintes maneiras.

18.4.2.1. O Evento ContainerInitialized

Devido ao poder do modelo de eventos typesafe de CDI, desenvolvedores de aplicação não precisam escrever qualquer código de inicialização. O módulo Weld SE vem com um método main embutido que inicializa o CDI para você e então dispara um evento `ContainerInitialized`. O ponto de entrada para o código de sua aplicação seria, portanto, um simples bean que observa o evento `ContainerInitialized`, como no exemplo anterior.

Neste caso sua aplicação pode ser iniciada chamando o método main fornecido como este:

```
java org.jboss.weld.environment.se.StartMain <args
>
```

18.4.2.2. API de Inicialização Programática

Para adicionar flexibilidade, CDI SE também vem com uma API de inicialização que pode ser chamada dentro de sua aplicação para inicializar a CDI e obter referências para os beans e eventos de sua aplicação. A API consiste em duas classes: `Weld` e `WeldContainer`.

```
public class Weld
{

   /** Boots Weld and creates and returns a WeldContainer instance, through which
    * beans and events can be accesed. */
   public WeldContainer initialize() {...}
```

```
   /** Convenience method for shutting down the container. */
   public void shutdown() {...}

}
```

```
public class WeldContainer
{

   /** Provides access to all beans within the application. */
   public Instance<Object
> instance() {...}

   /** Provides access to all events within the application. */
   public Event<Object
> event() {...}

   /** Provides direct access to the BeanManager. */
   public BeanManager getBeanManager() {...}

}
```

Aqui está um método main da aplicação de exemplo que usa esta API para inicializar um bean do tipo `MyApplicationBean`.

```
public static void main(String[] args) {
   WeldContainer weld = new Weld().initialize();
   weld.instance().select(MyApplicationBean.class).get();
   weld.shutdown();
}
```

Alternativamente a aplicação poderia ser iniciada ao disparar um evento personalizado que, então, seria observado por um outro simples bean. O seguinte exemplo dispara `MyEvent` na inicialização.

```
public static void main(String[] args) {
   WeldContainer weld = new Weld().initialize();
   weld.event().select(MyEvent.class).fire( new MyEvent() );
   weld.shutdown();
}
```

18.4.3. Thread Context

Em contraste com aplicações Java EE, aplicações Java SE coloca nenhuma restrição aos desenvolvedores a respeito da criação e uso de threads. Por isso o Weld SE fornece uma anotação de escopo específico, `@ThreadScoped`, e uma correspondente implementação de contexto que pode ser usada para vincular instâncias de beans à thread corrente. É destinada a ser usada em cenários onde você poderia usar `ThreadLocal`, e de fato fará uso de `ThreadLocal`, mas por baixo dos panos.

Para usar a anotação @ThreadScoped você precisa habilitar o `RunnableDecorator` que 'ouve' todas as execuções de `Runnable.run()` e decora elas configurando o contexto de thread antecipadamente, delimitando a thread corrente e destruindo o contexto mais tarde.

```
<beans>
  <decorators>
     <decorator
>org.jboss.weld.environment.se.threading.RunnableDecorator</decorator>
  </decorator>
</beans
>
```

Nota

Não é necessário usar @ThreadScoped em todas aplicações multithreaded. O contexto de thread não é destinado como um substituição a definir seus próprios cotextos específicos da aplicação. Somente é geralmente útil em situações onde você teria utilizado ThreadLocal diretamente, as quais são tipicamente raras.

18.4.4. Configurando o Classpath

O Weld SE vem empacotado como um jar 'iluminado' que inclui a API CDI, o Weld Core e todas classes dependentes agrupadas em um único jar. Por esse motivo o único jar do Weld que você precisa no classpath, em adição às classes e jars dependentes de sua aplicação, é o jar do Weld SE. Se você está trabalhando com uma aplicação Java SE pura você lançará ela usando `java` e isto pode ser mais simples para você.

Se você prefere trabalhar com dependências individuais, então você pode usar o jar `weld-core` que contém apenas as classes do Weld SE. Certamente neste modo você precisará montar o classpath você mesmo. Este modo é útil, por exemplo, se você deseja usar um slf4j alternativo.

Se você trabalha com uma solução de gerenciamento de dependências como o Maven, você pode declarar uma dependência com `org.jboss.weld.se:weld-se-core`.

Gerenciamento de contextos

19.1. Gerenciamento dos contextos embutidos

Weld permite que você gerencie facilmente os contextos embutidos através de injeção e execução dos métodos de ciclo de vida. Weld define dois tipos de contexto, *gerenciado* e *não gerenciado*. Contextos gerenciados podem ser ativados (permitindo instâncias de beans serem buscadas do contexto), invalidados (agendando instâncias de beans para serem destruídas) e desativados (evitando instâncias de beans de serem buscadas e, se o contexto foi invalidado, causando a destruição das mesmas. Contextos não gerenciados são sempre ativos, alguns podem oferecer a habilidade de destruir instâncias.

Os contextos gerenciados podem tanto ser *acoplados* ou *desacoplados*. Um contexto desacoplado possui um escopo dentro da thread em que foi ativado (instâncias colocadas em contexto numa thread não são visíveis nas outras threads), e é destruído após sua invalidação e desativação. Contextos acoplados são anexados a algum recipiente de dados externo (como o Http Session ou um mapa manualmente propagado) ao *associar* o recipiente com o contexto antes de ativar e o desassociando após desativar.

Dica

O Weld controla automaticamente o ciclo de vida do contexto em muitos cenários como em requisições HTTP, invocações remotas de EJB e invocações de MDB. Muitas das extensões de CDI oferece um ciclo de vida do contexto para outros ambientes, vale a pena verificar se existe uma extensão adequada antes de decidir gerenciar algum contexto por conta própria.

Weld provides a number of built in contexts, which are shown in *Tabela 19.1, "Contextos avaliáveis no Weld"*.

Tabela 19.1. Contextos avaliáveis no Weld

Escopo	Qualificadores	Contexto	Notas
`@Dependent`	`@Default`	`DependentContext`	O contexto dependente é desacoplado e não gerenciado
`@RequestScoped`	`@Unbound`	`RequestContext`	Um contexto de requisição desacoplado, útil para testes
`@RequestScoped`	`@Bound`	`RequestContext`	Um contexto de requisição acoplado acoplado a um mapa manualmente propagado, útil para testes ou em ambientes non-Servlet
	`@Default`	`BoundRequestContext`	
`@RequestScoped`	`@Http`	`RequestContext`	Um contexto de requisição acoplado a uma requisição Servlet, usado em qualquer contexto de requisição com base em Servlets
	`@Default`	`HttpRequestContext`	
`@RequestScoped`	`@Ejb`	`RequestContext`	Um contexto de requisição acoplado a um contexto

Escopo	Qualificadores	Contexto	Notas
	`@Default`	`EjbRequestContext`	de invocação de um interceptador, utilizado para invocações de EJB fora das requisições de Servlets
`@ConversationScoped`	`@Bound`	`ConversationContext`	Um contexto de conversação acoplado a dois mapas manualmente propagados (um que representa a requisição e outro que representa a sessão), útil para testes ou em ambientes non-Servlet
	`@Default`	`BoundConversationContext`	
`@ConversationScoped`	`@Http`	`ConversationContext`	Um contexto de conversação acoplado a uma requisição Servlet, usado em qualquer contexto de conversação com base em Servlets
	`@Default`	`HttpConversationContext`	
`@SessionScoped`	`@Bound`	`SessionContext`	Um contexto de sessão acoplado a um mapa manualmente propagado, útil para testes ou em ambientes non-Servlet
	`@Default`	`BoundSessionContext`	
`@SessionScoped`	`@Http`	`SessionContext`	Um contexto de sessão acoplado a uma requisição Servlet, usado em qualquer contexto de sessão com base em Servlets
	`@Default`	`HttpSessionContext`	
`@ApplicationScoped`	`@Default`	`ApplicationContext`	Um contexto de aplicação apoiado por um singleton com escopo de aplicação, é não-gerenciado e desacoplado mas oferece uma opção para destruir todas as entradas
`@SingletonScoped`	`@Default`	`SingletonContext`	Um contexto singleton apoiado por um singleton com escopo de aplicação, é não-gerenciado e desacoplado mas oferece uma opção para destruir todas as entradas

Os contextos não-gerenciados possuem pouco interesse em uma discussão sobre gerenciamento do ciclo de vida de contextos, assim, a partir daqui concentraremos nos contextos gerenciados (contextos não-gerenciados certamente desempenham um papel vital no funcionamento de sua aplicação e no Weld!). Como você pode observar na tabela acima, os contextos gerenciados oferecem uma série de diferentes implementações para o mesmo

escopo; em general, cada opção de contexto para um escopo possui a mesma API. Nós vamos ver abaixo uma série de cenários comuns no gerenciamento de ciclo de vida; de posse deste conhecimento e o Javadoc, você deverá ser capaz de lidar com qualquer das implementações de contexto que o Weld dispõe.

Vamos começar com o simples `BoundRequestContext`, o qual você pode usar para fornecer ao escopo de requisição fora de uma requisição Servlet ou uma invocação de EJB.

```
   /* Inject the BoundRequestContext. */
   /* Alternatively, you could look this up from the BeanManager */
   @Inject BoundRequestContext requestContext;

   ...

   /* Start the request, providing a data store which will last the lifetime of the request */
   public void startRequest(Map<String, Object
> requestDataStore) {
      // Associate the store with the context and acticate the context
      requestContext.associate(requestDataStore);
      requestContext.activate();
   }

   /* End the request, providing the same data store as was used to start the request */
   public void endRequest(Map<String, Object
> requestDataStore) {
      try {
         /* Invalidate the request (all bean instances will be scheduled for destruction) */
         requestContext.invalidate();
         /* Deactivate the request, causing all bean instances to be destroyed (as the context
 is invalid) */
         requestContext.deactivate();
      } finally {
         /* Ensure that whatever happens we dissociate to prevent any memory leaks */
         requestContext.dissociate(requestDataStore);
      }
   }
```

O contexto de sessão acoplado funciona da mesma maneira, exceto que a invalidação e desativação do contexto de sessão faz com que todas conversações na sessão sejam destruídas também. Os contextos de sessão e requisição http também funcionam de forma semelhante e podem ser úteis se você estiver criando threads a partir de uma requisição http). O contexto de sessão http oferece adicionalmente um método que pode destruir imediatamente o contexto.

Nota

Os contextos de sessão do Weld são "preguiçosos" e não requerem uma sessão para realmente existir até que uma instância de bean precise ser criada.

O contexto de conversação oferece mais algumas opções, as quais veremos por aqui.

```
   @Inject BoundConversationContext conversationContext;

   ...
```

```
   /* Start a transient conversation */
   /* Provide a data store which will last the lifetime of the request */
   /* and one that will last the lifetime of the session */
   public void startTransientConversation(Map<String, Object
> requestDataStore,
                                          Map<String, Object
> sessionDataStore) {
      resumeOrStartConversation(requestDataStore, sessionDataStore, null);
   }

   /* Start a transient conversation (if cid is null) or resume a non-transient */
   /* conversation. Provide a data store which will last the lifetime of the request */
   /* and one that will last the lifetime of the session */
   public void resumeOrStartConversation(Map<String, Object
> requestDataStore,
                                         Map<String, Object
> sessionDataStore,
                                         String cid) {
      /* Associate the stores with the context and acticate the context */
      * BoundRequest just wraps the two datastores */
      conversationContext.associate(new MutableBoundRequest(requestDataStore, sessionDataStore));
      // Pass the cid in
      conversationContext.activate(cid);
   }

   /* End the conversations, providing the same data store as was used to start */
   /* the request. Any transient conversations will be destroyed, any newly-promoted */
   /* conversations will be placed into the session */
   public void endOrPassivateConversation(Map<String, Object
> requestDataStore,
                                          Map<String, Object
> sessionDataStore) {
      try {
          /* Invalidate the conversation (all transient conversations will be scheduled for
 destruction) */
         conversationContext.invalidate();
        /* Deactivate the conversation, causing all transient conversations to be destroyed */
         conversationContext.deactivate();
      } finally {
        /* Ensure that whatever happens we dissociate to prevent memory leaks*/
         conversationContext.dissociate(new MutableBoundRequest(requestDataStore, sessionDataStore));
      }
   }
```

O contexto de conversação também oferece uma série de propriedades que controlam o comportamento da expiração da conversação (depois este período de inatividade a conversação será terminada e destruída pelo contêiner) e a duração do tempo limite das travas (o contexto de conversação assegura que uma única thread está acessando qualquer instância de beans ao travar o acesso, se uma trava não pode ser obtida depois de um certo tempo, o Weld lançará um erro em vez de continuar aguardando o travamento). Adicionalmente, você pode alterar o nome do parâmetro utilizado para transferir o id da conversação (por padrão, `cid`).

O Weld também introduz a noção de um `ManagedConversation`, a qual estende a interface `Conversation` com a habilidade de travar, destravar e afetar (atualizar o último timestamp utilizado) uma conversação. Finalmente, todas as conversações não-transientes em uma sessão podem ser obtidas a partir do contexto de conversação, assim como a conversação corrente.

Nota

As conversações do Weld não possuem ids atribuídos até que elas se tornem não-transientes.

Configuração

20.1. Evitando classes de serem escaneadas e implantadas

Weld permite que você evite que classes em seu arquivo sejam escaneadas, que acionem eventos de ciclo de vida, e que sejam implantadas como beans.

Neste tutorial iremos explorar essa funcionalidade através de um exemplo, uma especificação mais formal pode ser encontrada no xsd: *http://jboss.org/schema/weld/beans_1_1.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:weld="http://jboss.org/schema/weld/beans"
       xsi:schemaLocation="
          http://java.sun.com/xml/ns/javaee http://docs.jboss.org/cdi/beans_1_0.xsd
          http://jboss.org/schema/weld/beans http://jboss.org/schema/weld/beans_1_1.xsd">

    <weld:scan>

        <!-- Don't deploy the classes for the swing app! -->
        <weld:exclude name="com.acme.swing.**" />

        <!-- Don't include GWT support if GWT is not installed -->
        <weld:exclude name="com.acme.gwt.**">
            <weld:if-class-available name="!com.google.GWT"/>
        </weld:exclude>

        <!--
           Exclude classes which end in Blether if the system property verbosity is set to low
            i.e.
              java ... -Dverbosity=low
        -->
        <weld:exclude pattern="^(.*)Blether$">
            <weld:if-system-property name="verbosity" value="low"/>
        </weld:exclude>

       <!--
            Don't include JSF support if Wicket classes are present, and the viewlayer system
            property is not set
        -->
        <weld:exclude name="com.acme.jsf.**">
            <weld:if-class-available name="org.apahce.wicket.Wicket"/>
            <weld:if-system-property name="!viewlayer"/>
        </weld:exclude>
    </weld:scan>

</beans
>
```

Neste exemplo iremos mostrar o mais comum dos casos quando é preciso ter um maior controle de quais classes o Weld irá scanear. O primeiro filtro exclui todas as classes pertencentes a package `com.acme.swing`, e na maioria dos casos isso já será o suficiente para as suas necessidades.

Entretanto, as vezes é útil ser capas de ativar filtros dependendo do ambiente utilizado. Neste caso, Weld permite que você ative (ou desative) um filtro baseado em se tanto propriedades do sistema ou classes estão presente. O segundo filtro mostra o caso quando o escaneamento de certas classes é desativado dependendo do ambiente em que o sistema é implantado – neste caso estamos excluindo suporte ao GWT se o GWT não esta instalado.

Nota

Repare que usamos `!` no atributo `name` para inverter a condição de ativação do filtro. Você pode inverter qualquer condição de ativação deste modo.

O terceiro filtro usa uma expressão regular para selecionar as classes (normalmente utilizamos padrões simples baseados em nomes, eles não requerem que determinemos o ponto que delimita a hierarquia de pacotes).

Nota

Se você especificar somente o nome da propriedade do sistema, o Weld irá ativar o filtro se a propriedade tiver sido configurada (com qualquer valor). Se você também especificar o valor da propriedade do sistema, então o Weld irá ativar o filtro somente se o valor da propriedade corresponde exatamente com o valor dado a propriedade

O quarto filtro mostra uma configuração mais avançada, onde usamos múltiplas condições para decidir se filtro vai ser ativado ou não.

Você pode combinar quantas condições quiser (*todas* necessariamente tem que ser verdadeiras para o filtro ser ativado). Se você quiser que um filtro seja ativado se *qualquer* das condições seja verdadeira, você precisa de múltiplos filtros idênticos, cada um com condições de ativação diferentes.

Dica

No geral, a semântica definida pelas patterns sets do Ant (http://ant.apache.org/manual/dirtasks.html#patternset) são seguidas.

Apêndice A. Integrando o Weld em outros ambientes

Se você quer usar o Weld em um outro ambiente, você precisará fornecer certas informações ao Weld por meio da integração SPI. neste Apêndice nós discutiremos brevemente os passos necessários.

Serviços Corporativos

Se você quer somente utilizar beans gerenciados, e não quer tirar vantagem dos serviços corporativos (injeção de recursos EE, injeção CDI em classes de componente EE, eventos transacionais, suporte a serviços CDI nos EJBs) e implantações non-flat, então o suporte genérico por servlet provido pela extensão "Weld: Servlets" será suficiente e funcionará em qualquer contêiner que suporte a Servlet API.

Todos SPIs e APIs descritas possuem um extensivo JavaDoc, o qual explicita o contrato detalhado entre o contêiner e o Weld.

A.1. A SPI do Weld

A Weld SPI está localizada no módulo `weld-spi`, e empacotado como `weld-spi.jar`. Algumas SPIs são opcionais e deveriam ser implementadas somente se você precisar de substituir o comportamento padrão; outras são obrigatórias.

Todas as interfaces na SPI suportam o padrão decorador e fornecem a classe `Forwarding` localizada no sub-pacote `helpers`. Adicionalmente, as comumente utilizadas classes utilitárias e implementações padrão, também estão localizadas no sub-pacote `helpers`.

O Weld suporta múltiplos ambientes. Um ambiente é definido por uma implementação da interface `Environment`. Uma série de ambientes padrões já estão embutidos e descritos pela enumeração `Environments`. Os diferentes ambientes requerem diferentes serviços presentes (por exemplo um contêiner Servlet não requer transação, EJB ou serviços JPA). Por padrão um ambiente EE é assumido, mas você pode ajustar o ambiente chamando `bootstrap.setEnvironment()`.

O Weld utiliza um registro de serviço com tipagem genérica para permitir que os serviços sejam registrados. Todos os serviços implementam a interface `Service`. O registro de serviço permite que os serviços sejam acrescentados e recuperados.

A.1.1. Estrutura de implantação

Uma aplicação é normalmente composta por uma série de módulos. Por exemplo, uma aplicação Java EE pode conter vários módulos EJB (contendo lógica de negócio) e módulos war (contendo a interface de usuário). Um contêiner pode obrigar certas regras de *acessibilidade* como limitar a visibilidade de classes entre módulos. CDI permite que estas mesmas regras sejam aplicadas a resolução de beans e métodos observadores. Como as regras de acessibilidade variam entre contêineres, o Weld requer que o contêiner *descreva* a estrutura de implantação, por meio da SPI `Deployment`.

A especificação CDI aborda os *Bean Deployment Archives* (BDAs)—arquivos que são marcados como possuindo beans que devem ser implantados no contêiner CDI, e os tornam disponíveis para injeção e resolução. O Weld reusa esta descrição de *Bean Deployment Archives* em sua estrutura SPI de implantação. Cada implantaçao expõe

os BDAs que ela contém; cada BDA pode também referenciar outros que ele pode acessar. Conjuntamente, o percurso transitivo deste grafo forma os beans que são implantados na aplicação.

Para descrever a estrutura de implantação para o Weld, o contêiner deve fornecer uma implementação de `Deployment`. O método `Deployment.getBeanDeploymentArchives()` permite que o Weld descubra os módulos que compõem a aplicação. A especificação CDI também permite que beans sejam especificado programaticamente como parte da implantação do bean. Estes beans podem, ou não, estar em um BDA existente. Por esta razão, o Weld chamará `Deployment.loadBeanDeploymentArchive(Class clazz)` para cada bean descrito programaticamente.

Como os beans programaticamente descritos podem resultar em BDAs adicionais sendo inseridos ao grafo, o Weld descobrirá a estrutura BDA cada vez que um BDA desconhecido for retornado por `Deployment.loadBeanDeploymentArchive`.

BDAs Virtuais

Em um contêiner rigoroso, cada BDA deve especificar explicitamente quais outros BDAs ele pode acessar. Entretanto, muitos contêineres permitirão um mecanismo fácil para tornar os BDAs bi-direcionalmente acessíveis (como em um diretório de bibliotecas). Neste caso, é admissível (e razoável) descrever todos tais arquivos como um único e 'virtual' `BeanDeploymentArchive`.

Um contêiner, pode, por exemplo, usar uma estrutura de acessibilidade rasa para a aplicação. Neste caso, um único `BeanDeploymentArchive` deveria ser anexado ao `Deployment`.

O `BeanDeploymentArchive` fornece três métodos que permitem que seu conteúdo seja descoberto pelo Weld—`BeanDeploymentArchive.getBeanClasses()` deve retornar todas as classes no BDA, `BeanDeploymentArchive.getBeansXml()` deve retornar uma estrutura de dados representando o descritor `beans.xml` de implantação para o archive, e `BeanDeploymentArchive.getEjbs()` deve fornecer um descritor EJB para cada EJB no BDA, ou uma lista vazia se ele não for um arquivo EJB.

Para auxiliar o integrador do contêiner, o Weld fornece um parser do `beans.xml` embutido. Para analisar um `beans.xml` para a estrutura de dados requerida pelo `BeanDeploymentArchive`, o contêiner deve chamar `Bootstrap.parseBeansXml(URL)`. O Weld também pode analisar vários arquivos `beans.xml`, mesclando-os para se tornar uma única estrutura de dados. Isto pode ser realizado chamando `Bootstrap.parseBeansXml(Iterable<URL>)`.

Um BDA X também pode referenciar um outro BDA Y cujos beans podem ser resolvidos e injetados dentro de qualquer bean em BDA X. Estes são os BDAs acessíveis, e todo BDA que é diretamente acessível pelo BDA X deve ser retornado. Um BDA também terá BDAs que são transitivamente acessíveis, e o percurso transitivo do sub-grafo de BDA X descreve todos os beans resolvíveis pelo BDA X.

Correspondendo a estrutura do classloader com a implantação

Na prática, você pode respeitar a estrutura de implantação representada por `Deployment`, e o grafo virtual de BDAs será como um espelho da estrutura do classloader para uma implantação. Se uma classe pode a partir do BDA X ser carregada por outro no BDA Y, ele é acessível, e portanto os BDAs acessíveis do BDA Y deve incluir o BDA X.

Para especificar os BDAs diretamente acessíveis, o contêiner deve fornecer uma implementação de `BeanDeploymentArchive.getBeanDeploymentArchives()`.

Nota

O Weld permite ao contêiner descrever um grafo circular e converter um grafo para uma árvore como parte do processo de implantação.

Certos serviços são fornecidos para a implantação inteira, embora alguns sejam providos por-BDA. Os serviços BDA são fornecidos utilizando `BeanDeploymentArchive.getServices()` e somente aplicados aos BDAs que eles são providos.

O contrato de `Deployment` requer ao contêiner especificar as extensões portáveis (veja o capítulo 12 da especificação CDI) que deveriam ser carregadas pela aplicação. Para auxiliar o integrador do contêiner, o Weld fornece o método `Bootstrap.loadExtensions(ClassLoader)` que carregará as extensões para o classloader especificado.

A.1.2. Descritores EJB

O Weld delega a descoberta de beans do EJB 3 ao contêiner, uma vez que ele não duplica o trabalho feito pelo contêiner EJB e respeita qualquer extensão do fornecedor para a definição de EJB.

O `EjbDescriptor` deve retornar os metadados relevantes conforme definido na especificação EJB. Cada interface de negócio de um session bean deve ser descrita usando um `BusinessInterfaceDescriptor`.

A.1.3. Injeção de recursos EE e serviços de resolução

Todos os serviços de recursos EE são serviços por-BDA, e podem ser providos utilizando um de dois métodos. Qual método utilizar está a critério do integrador.

The integrator may choose to provide all EE resource injection services themselves, using another library or framework. In this case the integrator should use the EE environment, and implement the *Seção A.1.8, "Serviços de Injeção"* SPI.

Alternatively, the integrator may choose to use CDI to provide EE resource injection. In this case, the `EE_INJECT` environment should be used, and the integrator should implement the *Seção A.1.4, "Serviços EJB" [146]*, *Seção A.1.7, "Serviços de Recursos"* and *Seção A.1.5, "Serviços JPA"*.

Importante

CDI only provides annotation-based EE resource injection; if you wish to provide deployment descriptor (e.g. `ejb-jar.xml`) injection, you must use *Seção A.1.8, "Serviços de Injeção"*.

Se o contêiner realiza injeção de recursos EE, os recursos injetados devem ser serializáveis. Se a injeção de recursos EE for fornecida pelo Weld, o recurso resolvido deve ser serializável.

Dica

Se você usa um ambiente não-EE, então você pode implementar qualquer uma das SPIs de serviço EE, e o Weld proverá a funcionalidade associada. Não existe necessidade de implementar aqueles serviços que você não utilizará!

A.1.4. Serviços EJB

Os serviços EJB são separados em duas interfaces que são ambas por-BDA.

`EJBServices` é utilizado para resolver EJBs locais usados para apoiar session beans e sempre deve ser fornecido em um ambiente EE. `EJBServices.resolveEjb(EjbDescriptor ejbDescriptor)` retornar um envólucro—`SessionObjectReference`—sobre a referência do EJB. Este envólucro permite que o Weld solicite um referência que implementa uma dada interface de negócio, e, no caso de SFSBs, também solicitar a remoção do EJB do contêiner e consultar se o EJB tinha sido anteriormente removido.

`EJBResolutionServices.resolveEjb(InjectionPoint ij)` allows the resolution of `@EJB` (for injection into managed beans). This service is not required if the implementation of *Seção A.1.8, "Serviços de Injeção"* takes care of `@EJB` injection.

A.1.5. Serviços JPA

Assim como a resolução de EJB é delegada ao contêiner, a resolução de `@PersistenceContext` para injeção dentro dos beans gerenciados (com o `InjectionPoint` fornecido) é delegada ao contêiner.

To allow JPA integration, the `JpaServices` interface should be implemented. This service is not required if the implementation of *Seção A.1.8, "Serviços de Injeção"* takes care of `@PersistenceContext` injection.

A.1.6. Servicos de transação

O Weld delega as atividades JTA para o contêiner. A SPI fornece vários ganchos para facilmente conseguir isso com a interface `TransactionServices`.

Qualquer implementação de `javax.transaction.Synchronization` pode ser passada para o método `registerSynchronization()` e a implementação SPI deve registrar imediatamente a sincronização com o gerenciador de transação JTA usado pelos EJBs.

Para tornar mais fácil determinar se uma transação está ou não atualmente ativa para a thread solicitante, o método `isTransactionActive()` pode ser usado. A implementação SPI deve consultar o mesmo gerenciador de transação JTA usado pelos EJBs.

A.1.7. Serviços de Recursos

The resolution of `@Resource` (for injection into managed beans) is delegated to the container. You must provide an implementation of `ResourceServices` which provides these operations. This service is not required if the implementation of *Seção A.1.8, "Serviços de Injeção"* takes care of `@Resource` injection.

A.1.8. Serviços de Injeção

Um integrador pode desejar usar `InjectionServices` para fornecer injeção por campo ou método melhor do que o fornecido pelo Weld. Uma integração em um ambiente Java EE pode utilizar `InjectionServices` para prover injeção de recursos EE para beans gerenciados.

`InjectionServices` fornece um contrato muito simples, o interceptador `InjectionServices.aroundInject(InjectionContext ic);` será chamado para cada instância que CDI injeta, se for uma instância contextual, ou não-contextual injetada por `InjectionTarget.inject()`.

O `InjectionContext` pode ser usado para descobrir informações adicionais sobre a injeção sendo realizada, incluindo o `target` sendo injetado. `ic.proceed()` deve ser chamado para realizar injeção no estilo CDI e chama os métodos inicializadores.

A.1.9. Serviços de Segurança

No intuito de obter um `Principal` representando a identidade do requisitante atual, o contêiner deve fornecer uma implementação de `SecurityServices`.

A.1.10. Serviços da Bean Validation

A fim de se obter o `ValidatorFactory` padrão para a implantação da aplicação, o contêiner deve fornecer uma implementação de `ValidationServices`.

A.1.11. Identificando o BDA sendo endereçado

Quando um cliente faz uma requisição a uma aplicação que utiliza o Weld, a requisição pode ser endereçada a qualquer dos BDAs na aplicação. Para permitir que o Weld sirva corretamente a requisição, ele precisa saber qual BDA a requisição está endereçada. Onde possível, o Weld fornecerá algum contexto, mas o uso deste pelo integrador é opcional.

Nota

A maioria das Servlet usam um classloader por war, isto pode prover um bom meio para identificar o BDA em uso pelas requisições web.

Quando o Weld precisa identificar o BDA, ele usará um destes serviços, dependendo do que está servindo a requisição:

`ServletServices.getBeanDeploymentArchive(ServletContext ctx)`
: Identificar o war em uso. O `ServletContext` é fornecido por contexto adicional.

A.1.12. O armazenador de beans

O Weld utiliza um mapa como estrutura para armazenar instâncias de beans - `org.jboss.weld.context.api.BeanStore`. Você pode achar `org.jboss.weld.context.api.helpers.ConcurrentHashMapBeanStore` útil.

A.1.13. O contexto de aplicação

O Weld espera que o Servidor de Aplicação ou outro contêiner forneça o armazenamento para cada contexto da aplicação. O `org.jboss.weld.context.api.BeanStore` deve ser implementado para prover um armazenamento em escopo de aplicação.

A.1.14. Inicialização e Encerramento

A interface `org.jboss.weld.bootstrap.api.Bootstrap` define a inicialização para o Weld, a implantação e validação de beans. Para iniciar o Weld, você deve criar uma instância de `org.jboss.weld.bootstrap.WeldBeansBootstrap` (que implementa `Boostrap`), indicar quais serviços usará, e então solicitar que o contêiner inicie.

Toda a inicialização é separada em fases, a inicialização do contêiner, implantação dos beans, validação dos beans, e desligamento. A inicialização criará um gerenciador, adicionará os contextos embutidos e examinará a estrutura de implantação. A implantação de beans implantará todos os beans (definidos usando anotações, programaticamente ou embutidos). A validação de beans validará todos os beans.

Para inicializar o contêiner, você chama `Bootstrap.startInitialization()`. Antes de chamar `startInitialization()`, você deve registrar todos os serviços requeridos pelo ambiente. Você pode fazer isto chamando, por exemplo, `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. Você também deve fornecer o armazenador de beans do contexto da aplicação.

Tendo chamado `startInitialization()`, o `Manager` de cada BDA pode ser obtido chamando `Bootstrap.getManager(BeanDeploymentArchive bda)`.

Para implantar os beans descobertos, chame `Bootstrap.deployBeans()`.

Para validar os beans implantados, chame `Bootstrap.validateBeans()`.

Para colocar o contêiner em um estado onde ele pode servir requisições, chame `Bootstrap.endInitialization()`.

Para encerrar o contêiner você chama `Bootstrap.shutdown()`. Isto permite que o contêiner realize qualquer operação de limpeza necessária.

A.1.15. Carregando recursos

O Weld precisa carregar classes e recursos a partir do classpath em vários momentos. Por padrão, eles são carregados a partir do ClassLoader do contexto da Thread se disponível, se não o mesmo classloader que foi usado para carregar o Weld, entretanto este pode não ser o correto para alguns embientes. Se este é caso, você pode implementar `org.jboss.weld.spi.ResourceLoader`.

A.2. O contrato com o container

Existe uma série de requisitos que o Weld coloca sobre o contêiner para o correto funcionamento que não se enquadram na implementação de APIs.

Isolamento de Classloader (Classloader isolation)
: Se você está integrando o Weld em um ambiente que suporta implantação de várias aplicações, você de deve habilitar, automaticamente ou por configuração, o isolamento do classloader para cada aplicação CDI.

Servlet
: Se você esta integrando o Weld em um ambiente Servlet você deve registrar `org.jboss.weld.servlet.WeldListener` como um Servlet listener, seja automaticamente ou por configuração, para cada aplicação CDI que utiliza Servlet.

  Você deve assegurar que `WeldListener.contextInitialized()` seja chamada depois dos beans serem completamente implantados (garantir que `Bootstrap.deployBeans()` tenha sido chamado).

JSF
: Se você está integrando o Weld em um ambiente JSF você deve registrar `org.jboss.weld.jsf.WeldPhaseListener` como um phase listener.

  Se você etá integrando o Weld em um ambiente JSF você deve registrar `org.jboss.weld.el.WeldELContextListener` como um *listener* do contexto EL.

  Se você está integrando o Weld em um ambiente JSF você deve registrar `org.jboss.weld.jsf.ConversationAwareViewHandler` como um manipulador de visão delegante.

  Se você está integrando o Weld em um ambiente JSF você deve obter o gerenciador de beans para o módulo e então chamar `BeanManager.wrapExpressionFactory()`, passando

`Application.getExpressionFactory()` como argumento. A fábrica de expressão envolvida deve ser usada em todas as avaliações de expressões EL realizadas por JSF nesta aplicação web.

Se você está integrando o Weld em um ambiente JSF você precisa obter o gerenciador de beans para o módulo e então chamar `BeanManager.getELResolver()`. O EL resolver retornado deve ser registrado com o JSF para esta aplicação web.

Dica

Existem uma série de meios que você pode obter o gerenciador de beans para o módulo. Você poderia chamar `Bootstrap.getManager()`, passando o BDA deste módulo. Alternativamente, você poderia usar a injeção em classes de componentes Java EE, ou pesquisar o gerenciador de beans em JNDI.

Se você está integrando o Weld em um ambiente JSF você deve registrar `org.jboss.weld.servlet.ConversationPropagationFilter` como um Servlet listener, seja automaticamente ou por configuração, para cada aplicação CDI que utiliza JSF. Este filtro pode ser registrado em qualquer implantações Servlet sem problemas.

Nota

Weld somente suporta JSF 1.2 e versões posteriores.

JSP

Se você está integrando o Weld em um ambiente JSP você deve registrar `org.jboss.weld.el.WeldELContextListener` como um listener do contexto EL.

Se você está integrando o Weld em um ambiente JSP você deve obter o gerenciador de beans para o módulo e então chamar `BeanManager.wrapExpressionFactory()`, passando `Application.getExpressionFactory()` como argumento. A fábrica de expressão envolvida deve ser usada em todas as avaliações de expressões EL realizadas por JSP.

Se você está integrando o Weld em um ambiente JSP você deve obter o gerenciador de beans para o módulo e então chamar `BeanManager.getELResolver()`. O EL resolver retornado deve ser registrado no JSP para esta aplicação web.

Dica

Existem uma série de meios que você pode obter o gerenciador de beans para o módulo. Você poderia chamar `Bootstrap.getManager()`, passando o BDA deste módulo. Alternativamente, você poderia usar a injeção em classes de componentes Java EE, ou pesquisar o gerenciador de beans em JNDI.

Interceptador de Session Bean

Se você está integrando o Weld em um ambiente EJB você deve registrar o método `aroundInvoke` de `org.jboss.weld.ejb.SessionBeanInterceptor` como um interceptador EJB around-invoke para todos EJBs na aplicação, seja automaticamente ou por configuração, para cada aplicação CDI que utiliza beans corporativos. Se você está rodando em um ambiente EJB 3.1, você deve registrar este como um interceptador around-timeout também.

Importante

Você deve registrar o `SessionBeanInterceptor` como o interceptador mais interno na pilha para todos EJBs.

O `weld-core.jar`

O Weld pode residir dentro de um classloader isolado ou em um classloader compartilhado. Se você escolher utilizar um classloader isolado, o padrão `SingletonProvider`, `IsolatedStaticSingletonProvider`, pode ser usado. Se você escolher utilizar um classloader compartilhado, então você precisará escolher outra estratégia.

Você pode fornecer sua própria implementação de `Singleton` e `SingletonProvider` e registrá-la utilizando `SingletonProvider.initialize(SingletonProvider provider)`.

O Weld também fornece uma implementação da estratégia por aplicação com Thread Context Classloader, por meio de `TCCLSingletonProvider`.

Vinculando o gerenciador em JNDI

Você deveria vincular o gerenciador de beans ao arquivo de implantação de beans dentro de JNDI em `java:comp/BeanManager`. O tipo deve ser `javax.enterprise.inject.spi.BeanManager`. Para obter o gerenciador de beans correto para o arquivo de implantação de beans, você pode chamar `bootstrap.getBeanManager(beanDeploymentArchive)`.

Realizando injeção CDI em classes de componente Java EE

A especificação CDI requer que o contêiner forneça injeção dentro de recursos não-contextuais para todas classes de componentes Java EE. O Weld delega esta responsabilidade ao contêiner. Isto pode ser alcançado utilizando o SPI de CDI `InjectionTarget` já definido. Além disso, você deve realizar esta operação sobre o gerenciador de beans correto para o arquivo de implantação de beans que contenha a classe de componente EE.

A especificação CDI também requer que um evento `ProcessInjectionTarget` seja disparado para cada classe de componente Java EE. Além disso, se um observador chamar `ProcessInjectionTarget.setInjectionTarget()` o contêiner deve usar o alvo de injeção *especificado* para realizar a injeção.

Para ajudar o integrador, o Weld fornece o método `WeldManager.fireProcessInjectionTarget()`, o qual retorna o `InjectionTarget` a ser utilizado.

```
// Fire ProcessInjectionTarget, returning the InjectionTarget
// to use
InjectionTarget it = weldBeanManager.fireProcessInjectionTarget(clazz);

// Per instance required, create the creational context
CreationalContext<?> cc = beanManager.createCreationalContext(null);

// Produce the instance, performing any constructor injection required
Object instance = it.produce();

// Perform injection and call initializers
it.inject(instance, cc);

// Call the post-construct callback
it.postConstruct(instance);
```

```
// Call the pre-destroy callback
it.preDestroy(instance);

// Clean up the instance
it.dispose();
cc.release();
```

O contêiner pode intercalar outras operações entre estas chamadas. Além disso, o integrador pode escolher implementar qualquer dessas chamadas de uma outra maneira, assumindo que o contrato seja cumprido.

Ao realizar injeções em EJBs você deve utilizar o SPI definido pelo Weld, `WeldManager`. Além disso, você deve realizar esta operação sobre o gerenciador de beans correto para o arquivo de implantação de beans que contenha o EJB.

```
// Obtain the EjbDescriptor for the EJB
// You may choose to use this utility method to get the descriptor
EjbDescriptor<?> ejbDescriptor = beanManager.getEjbDescriptor(ejbName);

// Get an the Bean object
Bean<?> bean = beanManager.getBean(ejbDescriptor);

// Create the injection target
InjectionTarget it = deploymentBeanManager.createInjectionTarget(ejbDescriptor);

// Per instance required, create the creational context
CreationalContext<?> cc = deploymentBeanManager.createCreationalContext(bean);

// Perform injection and call initializers
it.inject(instance, cc);

// You may choose to have CDI call the post construct and pre destroy
// lifecycle callbacks

// Call the post-construct callback
it.postConstruct(instance);

// Call the pre-destroy callback
it.preDestroy(instance);

// Clean up the instance
it.dispose();
cc.release();
```