argonavis
tecnologia e arte

CDI
contexts and dependency injection

Helder da Rocha

JAVA EE 7

Capítulo 3: CDI, Interceptadores e Bean validation

1 CDI

A especificação CDI define serviços que permitem mapear objetos a contextos de um ciclo de vida, serem injetados em componentes e associados a interceptadores e decoradores, e interagirem com outros componentes de forma fracamente acoplada através da observação e disparo de eventos assíncronos.

Objetos injetáveis são chamados de *beans*, e incluem EJBs, *managed beans* e recursos Java EE. Instâncias de beans que pertencem a contextos (instancias contextuais) podem ser injetados em outros objetos através do serviço de injeção de dependências.

Os serviços são configurados através de *metadados*. No bean, através de anotações. Na aplicação através de descritores XML.

CDI simplifica a construção de aplicações Java EE. A integração entre as camadas Web e serviços é facilitada. Componentes EJB podem ser usados automaticamente como *managed beans* do JSF. É possível também criar componentes customizados para interagir com frameworks de terceiros.

1.1 Características do CDI

1.1.1 Injeção de dependências (DI)

Injeção de Dependências (DI) é um padrão de design de aplicações orientadas a objeto que consiste em inverter o controle usual que é do próprio objeto localizar ou construir suas dependências para tê-las inseridas no contexto por um agente externo. DI remove do componente a responsabilidade de localizar e configurar suas dependências.

DI tornou-se popular inicialmente através de um framework de Java Enterprise que hoje é uma alternativa concorrente ao Java EE: o Spring.

Em Java EE a injeção de dependências é realizada através de anotações. Uma anotação **@EJB** antes da declaração de um atributo do tipo de um Session Bean, faz o sistema procurar, localizar um bean compatível, instanciá-lo e disponibilizá-lo no contexto onde foi chamado. **@Resource** permite injetar recursos (ex: fábricas de objetos, conexões de bancos de dados, filas de mensageria, etc.) e **@PersistenceContext** é usado para injetar contextos de persistência JPA (ex: EntityManager). DI injeta os componentes através do seu tipo, mas às vezes é necessário identificar uma instância específica através de propriedades.

1.1.2 Contextos

Contextos definem um *escopo* no qual estão disponíveis serviços injetados. Podem durar o tempo da execução de um método, ou uma requisição, ou uma sessão do usuário, ou mesmo um período arbitrário estabelecido pela aplicação ou enquanto o serviço estiver no ar. Contextos são largamente utilizados em aplicações Web e JSF.

1.1.3 Contextos e Injeção de Dependências

O CDI praticamente torna obsoletos os mecanismos anteriores para injeção de serviços, declaração de escopos e beans gerenciados *(@ManagedBean*), oferecendo um framework integrado para configuração desses mecanismos. Proporciona a comunicação entre camadas de aplicações Java EE permitindo que serviços sejam injetados nos componentes que os requisitam através de uma API mínima baseada em anotações e tipo dos componentes. CDI reduz consideravelmente o acoplamento entre essas camadas.

A implementação de referência do CDI é JBoss Weld e sua especificação está publicada no grupo de trabalho do JSR 346. CDI é padrão obrigatório em servidores de aplicação que suportam Java EE 7.

1.2 Design para injeção de dependências

A injeção de dependências (DI) é um padrão de design que contribui para diminuir o acoplamento entre um componente e suas dependências, já que transfere a responsabilidade por criar e instanciar a dependência para a camada que é responsável por criar o componente. Portanto, em vez de um DAO instanciar um *DataSource*, ele pode ser injetado automaticamente pela classe que o criou, ou pelo container. Em vez do componente persistente localizar um DAO via JNDI, ele disponibiliza uma referência ou método *setter*, e deixa que outra classe ou o container forneça um.

DI não depende de nenhum framework. É um padrão de design clássico para aplicações orientadas a objeto (também chamado de inversão de controle, e indireção) e pode ser implementado usando Java puro. Os frameworks fornecem meios adicionais de desacoplamento (ex: anotações, XML, micro-containers). Os exemplos abaixo mostram como o padrão DI pode ser aplicado.

Considere as seguintes interface e classe. Elas representam um objeto que deverá ser armazenado em um mecanismo de persistência.

```
public interface Biblioteca {
     void emprestar(Livro livro);
     void devolver(Livro livro);
}

public class BibliotecaImpl implements Biblioteca {
     private BibliotecaStorage dao; // uma dependência!

     public void emprestar(Livro livro) {
         livro.status(Livro.EMPRESTADO);
          dao.atualizarStatus(livro);
     }

     public void devolver(Livro livro) {
         livro.status(Livro.DISPONIVEL);
          dao.atualizarStatus(livro);
     }
}
```

A interface abaixo, que é uma dependência de *BibliotecaImpl*, representa o objeto que cuidará do armazenamento:

```
public interface BibliotecaStorage {
```

```
    void atualizarStatus(Livro livro);
}
```

A implementação da dependência não precisa saber dos detalhes de *como* uma biblioteca armazena os dados sobre livros. Diferentes implementações poderiam existir.

```
public class BibliotecaNoSql implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando banco NoSQL");
    }
}
public class BibliotecaRemote implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando dados via rede");
    }
}
public class BibliotecaJdbc implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando banco JDBC");
    }
}
```

Pode até existir uma implementação para testes:

```
public class BibliotecaTeste implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando na Mock database");
    }
}
```

Sem usar injeção de dependências, a própria classe *Biblioteca* teria que escolher e instanciar o mecanismo de storage explicitamente (usando new ou JNDI, por exemplo):

```
public class BibliotecaImpl implements Biblioteca {
    private BibliotecaStorage dao = new BibliotecaNoSql();
    ...
}
```

Para usar injeção de dependências, é preciso que a classe ofereça uma interface que permita que se passe a referência externamente. Pode ser um *construtor* que receba o *dao* como parâmetro; pode ser um método *setter*, por exemplo:

```
public class BibliotecaImpl implements Biblioteca {
    private BibliotecaStorage dao; // dependencia

    public void setBibliotecaStorage(BibliotecaStorage ref) {
        this.dao = ref;
    }
    ...
}
```

Desta maneira, uma classe de controle (que cria a Biblioteca) poderia também instanciar uma ou mais dependências e injetar na *Biblioteca* a dependência desejada, em tempo de execução:

```
public static void main(String[] args) {
     BibliotecaImpl biblioteca = new BibliotecaImpl(); // criando o componente
     BibliotecaStorage dao = new BibliotecaNoSql();    // criando a dependência
     biblioteca.setBibliotecaStorage(dao);  // injetando a dependencia no componente
}
```

Isto permite um *acoplamento menor*, já que a classe *Biblioteca* não precisa conhecer a implementação dos métodos de *storage*, e está acoplada apenas à interface. Também garante *extensibilidade*, já que novas formas de *storage* que forem inventadas no futuro poderiam ser usadas; e *flexibilidade*, permitindo que o método de storage seja trocado durante a execução.

Este design não é novidade. É um dos usos clássicos de interfaces, aparece em diversos padrões de design, e muitas vezes é o padrão de acoplamento em projetos. A diferença em relação a DI usando algum framework é que a ligação entre objetos é realizada através de metadados como anotações ou arquivos XML, em vez de Java.

1.1 Injeção de dependências com CDI

CDI oferece um *framework* que inclui o uso de injeção de dependências. O seu uso não é limitado apenas à servidores de aplicação e pode ser usado em aplicações *standalone*. Se o ambiente de execução estiver configurado para usar CDI, é preciso apenas criar um arquivo chamado *beans.xml* (não precisa ter conteúdo) e coloca-lo na pasta *META-INF* do *Classpath* da aplicação para sinalizar que CDI deve ser usado. Depois disso, o ambiente de execução será ativado e as anotações do CDI serão interpretadas.

Para injetar o serviço na classe *Biblioteca*, anotamos o método ou o atributo com *@Inject*.

```
@Inject public void setBibliotecaStorage(BibliotecaStorage ref) {
        this.dao = ref;
}
```

CDI identifica a instância pelo tipo, ou seja, pela classe. Se houver apenas uma implementação, ela será localizada e usada. Se houver mais de uma é preciso escolher uma que será a implementação default. Ela *não precisa de anotação* (é opcional – pode-se usar *@Default*):

```
@Default // anotação sempre opcional
public class BibliotecaNoSql implements BibliotecaStorage {
     public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando banco NoSQL");
```

```
    }
}
```

Outras implementações da interface, se existirem no classpath *precisam* ser anotadas com *@Alternative*:

```
@Alternative // anotação obrigatória havendo outras implementações desta interface
public class BibliotecaTeste implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
            System.out.println("Atualizando na Mock database");
    }
}
```

Se a biblioteca representar um *Model* em uma arquitetura MVC, ou se ela precisar ser acessada via expressões EL, ou se for armazenada no registro de um container, deve-se anota-la com *@Named*:

```
@Named("biblioteca")
public class BibliotecaImpl implements Biblioteca {}
```

O nome default é o próprio nome (não qualificado) da classe, de acordo com as regras de formação de JavaBeans. Sem a propriedade acima, o nome seria "*bibliotecaImpl*".

1.3 Como usar CDI

CDI é nativo a qualquer servidor Java EE 7. Caso o CDI seja usado de forma standalone, fora de um servidor, é preciso instalar uma implementação. JBoss Weld é a implementação de referência mas há outras implementações: Apache OpenWebBeans, Caucho CanDI.

1.3.1 Instalação em ambiente standalone

Para instalar, baixe o Weld em

```
http://seamframework.org/Weld/WeldDistributionDownloads
```

copie o *weld-servlet.jar* para a pasta *WEB-INF/lib*, ou use o Maven e importe as dependências da API e do Weld. Importe a API para poder compilar o código (e para ambiente standalone ou servidor que não suporta CDI:

```
<dependencies>
  <dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <version>1.1</version>
  </dependency>
...
</dependencies>
```

O pacote acima contém apenas a interface. Se o ambiente não suporta CDI, será preciso também incluir a implementação. Esta é a implementação de referência Weld:

```
<dependency>
  <groupId>org.jboss.weld.se</groupId>
  <artifactId>weld-se</artifactId>
  <version>2.2.6.Final</version>
</dependency>
```

Que requer a configuração de um *listener* no *web.xml*:

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

Se o CDI estiver sendo usado em um servidor de aplicações Java EE 7, esta etapa não é necessária.

1.3.2 Ativação

O próximo passo é criar um arquivo beans.xml (vazio) e armazená-lo na pasta META-INF dentro do classpath da aplicação. Em Java EE 7 o beans.xml não precisa conter nada, mas se contiver XML, deve conter pelo menos o elemento raiz com o seguinte código:

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="annotated">
</beans>
```

Isto garante que o servidor irá processar anotações CDI. Agora pode-se anotar beans, serviços, etc. com as anotações CDI.

1.3.3 Uso do CDI

Uma vez configurado, o uso do CDI envolve principalmente anotações:

- Usar a anotação *@Inject* para anotar um atributo (ou método) de uma classe como *dependência injetável*
- Usar (opcionalmente) *@Default* para anotar a implementação default da dependência que será injetada
- Se houver outras implementações, usar (obrigatoriamente) *@Alternative* para anotá-las.

- Usar *@Named* para dar um nome à implementação do serviço no registro CDI. Este nome permitirá que o bean seja acessível via EL (ex: em JSF)

1.3.4 Exemplo de uma aplicação usando CDI

Exemplo de um serviço (*MailService*) representado por uma implementação *@Default* (*TestMailService*) em escopo CDI (*@ApplicationScoped*):

```
@ApplicationScoped
public class TestMailService implements MailService {
   public enviar(String de, String para, String texto) {
       ... // realiza o envio do email
   }
}
```

Exemplo de um bean em escopo CDI *(@SessionScoped*) e disponível em registro EL com o nome "usuario":

```
@SessionScoped
@Named
 public class Usuario {
     public String getEmail() {
       ...
     }
     ...
}
```

Exemplo de um bean em escopo CDI *(@RequestScoped*), disponível em registro EL com o nome "mail" que injeta os dois beans anteriores.

```
@RequestScoped
@Named
public class Mail {
   private @Inject MailService servico;
   private @Inject Usuario usuario;

   private String mensagem, destinatario;

   public String getEmail() { ... }
   public String getDestinatario() { ... }
   public void setEmail(String email) { ... }
   public void setDestinatario(String destinatario) { ... }

   public String enviarEmail() {
      servico.enviar(usuario.getEmail(),"fulano@mail.com", "Olá");
      return "enviada";
   }
}
```

Exemplo de um formulário JSF usando os managed beans declarados acima através de seus nomes EL (definidos com **@Named**), acessando seus métodos e propriedades:

```
<h:form>
   <h:outputLabel value="Usuário" for="usuario"/>
     <h:outputText id="usuario" value="#{usuario.email}"/><br/>

   <h:outputLabel value="Destinatario" for="destinatario"/>
     <h:inputText id="destinatario" value="#{mail.destinatario}"/><br/>

   <h:outputLabel value="Body" for="body"/>
     <h:inputText id="body" value="#{mail.mensagem}"/><br/>

   <h:commandButton value="Send" action="#{mail.enviarEmail}"/>
</h:form>
```

1.4 JavaBeans, Enterprise JavaBeans e Managed Beans

Objetos injetáveis via CDI são JavaBeans, ou simplesmente Beans. Beans são objetos que seguem convenções mínimas de programação. Quase qualquer classe que tenha um construtor default (sem parâmetros) pode ser considerada um bean. CDI permite que beans sejam gerenciáveis (managed) pelo container.

A especificação CDI afirma que um JavaBean é qualquer classe Java que

- Não é classe interna (a menos que seja estática)
- Seu construtor não tem parâmetros ou é anotado com *@Inject*

Um Enterprise JavaBean (EJB) é um tipo de JavaBean que possui uma anotação que o qualifica como EJB *(@Remote*, *@Local*, *@Stateless*, etc.), ou é declarado como EJB em um arquivo de configuração ejb-jar.xml.

O CDI define um *Managed Bean* como qualquer JavaBean que não seja um Enterprise *JavaBean* e que:

- É uma classe concreta (ou anotada com *@Decorator*)
- Não implementa a interface Extension (javax.enterprise.inject.spi)

Na prática, tanto Session Beans quanto Managed Beans *são* beans gerenciáveis (ou seja, managed). Pode-se injetar um session bean em um managed bean e vice-versa. Message-driven beans são objetos não contextuais e não podem ser injetados em outros objetos. Tanto EJBs como Managed Beans suportam callbacks de ciclo de vida *@PostConstruct* e *@PreDestroy*.

1.4.1 Quando usar um Session Bean em vez de um Managed Bean?

No Java EE 7 é melhor usar um Session Bean quando houver necessidade de:

- Transações e segurança com granularidade de método
- Controle de concorrência em singletons
- Passivação de nível de instância para Stateful beans e *pooling* para Stateless
- Chamadas remotas ou web services
- Timers e métodos assíncronos

Embora possam ser usados em escopos longos, Managed Beans são mais fáceis de gerenciar em escopos de duração menor, como RequestScoped, ou com limites bem definidos (*ViewScoped*, *ConversationScoped*). Em geral, beans usados em contextos *SessionScoped* ou *ApplicationScoped* devem (preferencialmente) ser EJBs.

1.4.2 Injeção de Managed Beans

Um Managed Bean possui um *conjunto de tipos*, pelos quais pode ser identificado. Seus tipos incluem sua classe, superclasse e interfaces visíveis pelo cliente.

Um Managed Bean também tem uma coleção de *qualificadores*. Qualificadores são declarados como anotações. Todo bean possui pelo menos o qualificador *@Default*. Se um bean possui várias implementações, elas podem receber qualificadores diferentes para que possam ser selecionadas durante a injeção.

Todo managed bean pertence a um *escopo*, que determina a sua duração. O escopo default é *@Dependent* (herda o escopo da classe que o injetou)

O contrato que precisa ser satisfeito pelo bean que é injetado requer que seja possível distinguir uma implementação única através da combinação do seu *tipo* mais *qualificadores*.

1.5 Escopos do CDI

Todo bean em CDI existe em um escopo. Escopos podem ser declarados antes da classe. A maioria são semelhantes aos do JSF e têm o mesmo nome:

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

Se um bean não declara nenhum escopo, ele pertence ao escopo do objeto onde foi injetado. Isto corresponde ao pseudo-escopo **@Dependent**. É recomendado anotar um bean como *@Dependent* (mesmo sendo opcional) se esse comportamento for intencional. Os escopos listados acima são escopos normais (são meta-anotados, na sua definição, como *@NormalScope*). Outros frameworks podem usar *@NormalScope* para definir escopos adicionais. Alguns outros são definidos no framework JSF (*@ViewScoped, @FlowScoped*).

1.6 Qualificadores

CDI injeta tipos, e não instâncias. Se uma aplicação precisa manter de múltiplas implementações da mesma interface e elas têm o mesmo nome, não será possível selecionar ambas. Para isto podem definir um *@Qualifier*, que é uma anotação específica para qualificar diferentes implementações.

Por exemplo, se houver dois diferentes *EntityManagers*, pode-se definir um *@Qualifier* para cada um deles. Cada um será uma anotação, como esta:

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

Suponha que exista também outro qualificador *@AdminDb*. Agora os dois EntityManagers podem ser identificados:

```
public @ApplicationScoped class MyService {
 private @Inject @CustomerDb EntityManager customerEm;
 private @Inject @AdminDb EntityManager adminEm;
    …
```

Neste outro exemplo temos uma implementação diferente para o serviço de storage da biblioteca:

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

Uma vez definido, o qualificador pode ser usado para remover a ambiguidade do *tipo*:

```
@Inject @ServicoRemoto BibliotecaStorage storage;
```

O container vai procurar por beans to tipo PaymentProcessor que tenham qualificador @ServicoRemoto. Portanto é preciso anotar o bean com essa anotação:

```
@ServicoRemoto
public class BibliotecaRemote implements BibliotecaStorage {
    public void atualizarStatus(Livro livro) {
```

```
            System.out.println("Atualizando dados via rede");
    }
}
```

Qualquer bean ou ponto de injeção que não especificar um qualificador possui o qualificador default *@Default*.

1.7 Métodos produtores

Muitas vezes o objeto para ser injetado não pode ser simplesmente criado com *new*. Às vezes é necessário determinar uma implementação específica em tempo de execução. Às vezes o objeto é obtido através de um outro meio e precisa ser injetado. Para essas situações pode-se usar métodos produtores. Um método produtor deve retornar uma instância do objeto que produz e ser anotado com *@Produces*. Usando produtores é possível injetar praticamente qualquer coisa. Este exemplo (da documentação oficial do Weld) mostra como criar um produtor de inteiros aleatórios, que pode ser injetado em qualquer bean:

```
import javax.enterprise.inject.Produces;
@ApplicationScoped
public class RandomNumberGenerator {
    private java.util.Random random =
                            new java.util.Random(System.currentTimeMillis());
    @Produces @Named @Random int getRandomNumber() {
        return random.nextInt(100);
    }
}
```

Foi especificado um qualificador (para distinguir o inteiro aleatório de outros componentes que produzam Integer) e seu escopo é o default *@Dependent*, ou seja, herda o escopo do objeto onde é usado. Como foi usado *@Named*, o método também possui um *nome* para uso em EL: *randomNumber*. Agora pode-se injetar um número aleatório da forma:

```
@Inject @Random int numeroAleatorio;
```

Ou chamá-lo em uma expressão EL:

```
<p>Número da sorte: #{randomNumber}</p>
```

O produtor é um método concreto de uma classe de managed bean ou session bean.

Um método produtor pode conter parâmetros. Neste caso o container irá buscar um bean instanciado que combine com o tipo e qualificadores de cada parâmetro e injetar automaticamente. Se a instância não for encontrada, ou se houver mais de uma que combine com tipo e qualificador, uma exceção será lançada.

```
@Produces List<Livro> acervo(@Central Biblioteca biblioteca) {
    return biblioteca.getLivros();
}
```

Campos produtores podem ser usados anotando um atributo que declara e instancia uma variável diretamente:

```
@Produces Storage storage = new BDStorage();
```

O container escolhe o objeto a ser injetado com base no *tipo* do *ponto de injeção*.

2 Interceptadores

Interceptadores são operações que podem ser injetadas em outros métodos durante a chamada. Representam interesses ortogonais e são uma abstração da programação orientada a aspectos (AOP).

Um interceptador pode ser aplicado a vários métodos através de anotações, permitindo injetar operações adicionais em métodos que serão aplicados em pontos de junção: antes ou depois dos métodos interceptados.

2.1 Tipos de interceptadores

Interceptadores são callbacks especiais que são chamados quando acontecem eventos relacionados ao ciclo de vida e execução dos objetos. Existem cinco anotações. Duas delas são chamadas ou antes ou depois de um evento, e as outras três são chamadas em volta do evento.

- *@AroundConstruct* - Marca o método como um interceptador que recebe um callback depois que a classe-alvo e construida
- *@AroundInvoke* - Marca o metodo como um interceptador
- *@AroundTimeout* - Interceptador de timeout
- *@PostConstruct* - Interceptador para eventos PostConstruct
- *@PreDestroy* - Interceptador para eventos PreDestroy

Vimos aplicações de *@PostConstruct* e *@PostDestroy* em vários managed e enterprised beans. *@AroundInvoke* é talvez o mais usado. *@AroundTimeout* é usado em timers, e *@AroundConstruct* é usado raramente.

2.2 Como criar um interceptador @AroundInvoke

Para usar é preciso criar uma classe contendo um método anotado com *@AroundInvoke* recebendo parâmetro *InvocationContext*. A classe não precisa ter nenhuma anotação (*@Interceptor* é opcional):

```
@Interceptor
public class Intercep1 {
   @AroundInvoke
```

```
   public Object metodo(InvocationContext ctx) throws Exception {...}
}
```

Chame *invocationContext.proceed()* no ponto do código em que o método a ser interceptado deverá ser chamado:

```
@Interceptor
public class XmlLogger {
   @AroundInvoke
   public Object xmlLog(InvocationContext ctx) throws Exception {
      System.out.println("<log>");
      Object resultado = ctx.proceed();
      System.out.println("</log>");
      return resultado;
   }
}
```

Use a anotação *@Interceptors* nas classes cujos métodos deverão ser interceptados. A anotação pode receber uma, ou uma lista de interceptadores, para que todos os seus métodos sejam interceptados:

```
@Stateless
@Interceptors({XmlLogger.class, Intercept2.class, ...})
public class MyBean implements MyBeanIF { ... }
```

A declaração também pode ser feita em cada método.

2.3 Interceptor Binding

Também é possível configurar um interceptador usando *<interceptor-binding>* no ejb-jar.xml ou web.xml ou usado a anotação *@Priority* na classe do interceptador para associá-lo a um qualificador especial chamado de *@InterceptorBinding*. Desta forma é possível criar anotações que representam a injeção dos interceptadores.

Por exemplo, para em vez de usar *@Interceptors* nos métodos e classes poderíamos usar *@XmlLogger*, primeiro criando um qualificador:

```
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
@Inherited
pubic @interface XmlLogger { ... }
```

E usando essa anotação para anotar o interceptador da forma:

```
@XmlLogger
@Priority(Interceptor.Priority.APPLICATION)
public class XmlLogger {
   @AroundInvoke
   public Object xmlLog(InvocationContext ctx) throws Exception {...}
```

```
    ...
}
```

Assim, é possível anotar os métodos e classes com o interceptador usando apenas *@XmlLogger*:

```
@Stateless @XmlLogger
public class MyBean implements MyBeanIF { ... }
```

2.4 Aplicações para interceptadores

Aplicações típicas de interceptadores incluem loggers, segurança e transações. Várias anotações desse tipo existem em CDI e foram construídas com interceptadores. Por exemplo, uma anotação para inserir um contexto transacional pode ser feita com um interceptador, definindo um *@InterceptorBinding*:

```
@Retention(RetentionPolicy.RUNTIME)
@Target( { ElementType.TYPE, ElementType.METHOD })
@InterceptorBinding
public @interface ContextoTransacional {}
```

Depois o interceptador:

```
@ContextoTransacional
@Priority(Interceptor.Priority.APPLICATION)
public class ContextoTransacionalInterceptor {

   @AroundInvoke
   public Object xmlLog(InvocationContext ctx) throws Exception {
        Object resultado;
        try {
            ut.begin();
            resultado = ctx.proceed();
            ut.commit();
        } catch (Exception e) {
            ut.rollback();
        }
        return resultado;
   }
}
```

Agora o interceptador pode ser usado para tipos (classes) ou métodos:

```
@ApplicationScoped
public class FilmeService {
   private @Inject EntityManager em;

   @ContextoTransacional
   public gravarFilme(Filme f) {
      em.persist(f);
   }
}
```

3 Bean Validation

Bean Validation é uma coleção de anotações para configurar validação em beans. Essas validações são usadas automaticamente em frameworks como JSF. As anotações fazem parte do pacote *javax.validation.constraints* e representam restrições (*constraints*) que são aplicadas em classe, atributos, métodos, parâmetros e que provocam exceções se violadas.

3.1 Como usar

Para usar *Bean Validation* basta anotar uma propriedade, um método ou seus parâmetros com anotações. O sistema verificará se os valores são compatíveis com as restrições de validação e lançará uma exceção se não forem.

Por exemplo:

```
public class Name {
   @NotNull
   private String firstname;

   @NotNull
   private String lastname;

}
```

As anotações default da API de Bean Validation estão listadas abaixo:

`@AssertFalse`
`@AssertTrue`
`@NotNull`
`@Null`
	Propriedade deve ser true ou false, nula ou não

`@DecimalMax`
`@DecimalMin`
`@Max`
`@Min`
	Especifica um valor máximo ou mínimo

`@Digits(integer=n, fraction=m)`	Dígitos (ex: 12.345 = integer=2, fraction=3)
`@Future` `@Past`	Data tem que ser no futuro ou no passado
`@Pattern(regexp="pattern")`	Expressão regular deve combinar com o string
`@Size(min=n, max=m)`	Tamanho do string

Anotações podem ser combinadas:

```
public class Name {
```

```
    @NotNull
    @Size(min=1, max=16)
    private String firstname;

    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

Anotações também podem ser estendidas e customizadas.

4 Referências

[1] *JSR-346 Contexts and Dependency Injection for the Java EE platform (CDI)*. Red Hat. 2014. https://docs.jboss.org/cdi/spec/1.2/cdi-spec-1.2.pdf

[2] Bean Validation Expert Group. Emmanuel Bernard. *Bean Validation Specification 1.1*. Red Hat, 2013. http://beanvalidation.org/1.1/spec/

[3] Eric Jendrock et al. *The Java EE 7 Tutorial*. Oracle. 2014. https://docs.oracle.com/javaee/7/JEETT.pdf

[4] Linda deMichiel and Bill Shannon. *JSR 342. The Java Enterprise Edition Specification. Version 7*. Oracle, 2013. http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-spec/JavaEE_Platform_Spec.pdf

[5] Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014.