Lab 4 – CDI - Injeção de Dependência e Contextos

Neste laboratório.

Exercícios

Exercício 1: Configurando um projeto CDI.

Exercício 2: Injeção de Dependências

Exercício 3: Qualificadores

Exercício 4: Produtores

Exercício 5: Alternatives

Exercício 6: Interceptadores e Logger

Exercício 1 – Configurando um projeto CDI

1. Para adicionar o CDI a aplicação primeiro você deve fazer com que a aplicação seja compatível com o CDI. Primeiro, adicione as dependencias do Weld no **pom.xml** do seu projeto.

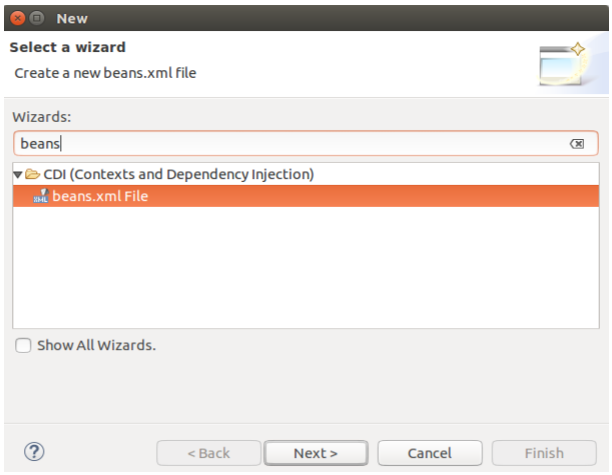
```
<!-- WELD CDI -->
      <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>1.2</version>
            <scope>provided</scope>
      </dependency>

      <dependency>
        <groupId>org.jboss.weld.servlet</groupId>
         <artifactId>weld-servlet</artifactId>
         <version>1.1.10.Final</version>
      </dependency>

<!-- Util Logger -->
      <dependency>
            <groupId>com.logentries</groupId>
            <artifactId>logentries-appender</artifactId>
            <version>1.1.29</version>
      </dependency>
```

   OBS: Essas dependências estão configuradas somente para o **Tomcat.** Se você estiver usando ou for usar outros servers como o JBoss ou o WildFly, a configuração deve ser outra.

2. Agora, dentro da pasta **webapp**, crie a pasta **META-INF** e dentro dela, crie o arquivo **beans.xml**, assim como mostrado abaixo. Vá em File → New → Other... e procure por beans.xml File. Clique em **Next**, selecione a pasta onde deseja salvar (META-INF) e por último, **Finish**.



3. Crie, na mesma pasta **META-INF**, o arquivo **context.xml** e adicione o código abaixo.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
	<!-- disables storage of session across restarts -->
	<Manager pathname=""/>
	<Resource name="BeanManager" auth="Container"
		type="javax.enterprise.inject.spi.BeanManager"
		factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context>
```

4. Feito isso, vá ao arquivo **web.xml** e adicione o código abaixo.

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

<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>
```

Finalizados todos esses passos, seu projeto já estará configurado para funcionar em CDI. Agora vamos fazer alguns exemplos.

Exercício 2 – Injeção de Dependências

1. Crie, dentro do pacote **controller** o arquivo **PrecoProdutoBean.xhtml** e nele, crie um objeto **getPreco()**, conforme o código abaixo.

```
package controller;

	@Named
	public class PrecoProdutoBean {

		public double getPreco(){
			return 12.55;
		}
	}
```

2. Agora, para testar e conferir se só com a anotação @Named sua classe é um Bean, crie o arquivo **PrecoProduto.xhtml** e imprima o preço na tela, adicionando o código a seguir.

```
<body>
	<h:outputText value="#{precoProdutoBean.preco}"/>
</body>
```

Teste sua página no navegador e veja o resultado. O número 12,55 deverá ser imprimido na tela.

3. Crie uma classe no pacote **service** com o nome de **CalculadoraPreco** e nela, coloque o seguinte código.

```
package service;

public class CalculadoraPreco {

	public double calcularPreco(int quantidade, double precoUnitario){
		return quantidade*precoUnitario;
	}
}
```

4. Para fazer uso dessa classe no nosso PrecoProdutoBean iremos fazer uma **injeção no campo**. Modifique sua classe **PrecoProdutoBean** para que fique igual ao código abaixo.

```
public class PrecoProdutoBean {

	@Inject
	private CalculadoraPreco calculadora;

	public double getPreco(){
		return calculadora.calcularPreco(12, 44.57);
	}
}
```

Teste novamente sua página. Verifique que sem a Annotation tag `@Inject`, a variável calculadora receberá um **NullPointerException**, pois ela não foi iniciada. Com essa tag, é feita a injeção da classe dentro do nosso bean, fazendo com que funcione.

5. Como vimos anteriormente, essa injeção pode ser feita em vários pontos, que são chamados de pontos de injeção. Acima, vemos a injeção no campo, vamos agora ver uma **injeção no construtor**. Crie outra classe similar a criada acima e faça as alterações necessárias para fazer o código abaixo funcionar na página **PrecoProduto.xhtml**.

```
public class PrecoProdutoBean3 {

	private CalculadoraPreco calculadora;

	@Inject
	public void PrecoProdutoBean2(CalculadoraPreco calculadora){
		this.calculadora = calculadora;
	}
}
```

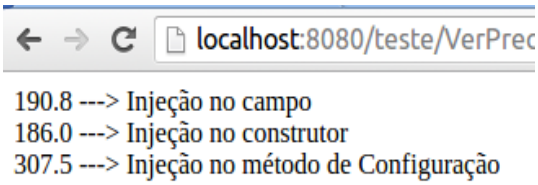
6. Agora, veremos o exemplo de ponto de **injeção no método de configuração.** Crie uma nova classe com os mesmos elementos da classe PrecoProdutoBean e faça a seguinte alteração.

```
public class PrecoProdutoBean3 {

	private CalculadoraPreco calculadora;

	@Inject
	public void setCalculadora(CalculadoraPreco calculadora){
		this.calculadora = calculadora;
	}
}
```

Faça os testes e veja os resultados.



Exercício 3 – Qualificadores

1. Vamos criar um mensageiro e a partir de uma interface, e pelos qualificadores, definir se a mensagem deve ser passada por correios ou por SMS. Crie dentro do pacote **service** uma interface **Mensageiro**, assim como no código abaixo

```
package service;

public interface Mensageiro {

	public void enviarMensagem(String mensagem);

}
```

2. No Pacote **controller**, crie o bean **EnviarMensagemBean** e implemente-o para que receba uma mensagem do servidor e envie essa mensagem ao nosso console do Eclipse.

```
package controller;

import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

import service.*;

@Named
@RequestScoped
public class EnvioMensagemBean {

	@Inject
	private Mensageiro mensageiro;

	private String texto;

	public void enviarMensagem(){
		mensageiro.enviarMensagem(texto);
	}

	public String getTexto() {
		return texto;
	}

	public void setTexto(String texto) {
		this.texto = texto;
	}
}
```

3. Crie um arquivo **EnviarMensagem.xhtml** para receber uma mensagem e passar para o bean do passo anterior. Segue o código abaixo.

```
<body>

	<h:form>
		<p:inputText value="#{envioMensagemBean.texto}"/>
		<p:commandButton value="Enviar Mensagem"
			action="#{envioMensagemBean.enviarMensagem}"/>
```

```
        </h:form>
    </body>
```

4. Até agora nada acontecerá, pois ainda não definimos o que o objeto enviarMensagem irá fazer com o texto que ele receber. Vamos criar no pacote service mais duas classes, a classe **MensageiroCorreio** e **MensageiroSMS** como está no código abaixo.

MensageiroCorreio.class

```
package service;

public class MensageiroCorreio implements Mensageiro{

	public void enviarMensagem(String mensagem){
		System.out.println("Enviando Mensagem por correio: "+mensagem);
	}

}
```

MensageiroSMS.class

```
package service;

public class MensageiroSMS implements Mensageiro{

	public void enviarMensagem(String mensagem) {
		System.out.println("Enviando mensagem por SMS: "+mensagem);
	}
}
```

5. Até esse passo a aplicação não funcionará, pois estamos adicionando dois métodos de saida diferentes sem passar para o bean qual que deverá ser usado, precisamos qualificá-los de acordo com a urgência de nossa mensagem. Vamos agora criar uma **Annotation** no nosso pacote **service**, chamada `@Urgente`. Código segue abaixo.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD,
	ElementType.PARAMETER })
public @interface Urgente {

}
```

6. Agora, adicione a tag `@Urgente` na classe **MensageiroSMS** como no código abaixo.

```
	package service;

	@Urgente
	public class MensageiroSMS implements Mensageiro{


	public void enviarMensagem(String mensagem) {
		System.out.println("Enviando mensagem por SMS: "+mensagem);
	}
}
```

7. Teste agora sua página, veja que sem colocar nada diretamente no seu **EnvioMensagemBean** ele já pega o método **MensageiroCorreio**. Isso é porque ele é o padrão, no momento. É como se na classe **MensageiroCorreio** tivesse a annotation tag `@Default` . Agora, para usar o **MensageiroSMS**, devemos colocar a tag `@Urgente` dentro do ManagedBean, assim como mostrado no código abaixo.

```
@Inject @Urgente
private Mensageiro mensageiro;
```

Teste as alterações, sempre prestando atenção no código.

Exercício 4 – Produtores

1. Com os produtores, podemos injetar qualquer objeto em qualquer lugar. Como exemplo, faremos um método de **Banco**. Crie uma interface **BancoService** com o método sacar().

```
public interface BancoService {

	void sacar(double valor);
}
```

2. Em seguida, crie uma classe **BancoServiceImpl** que será a implementação do nosso BancoService. Nele, vamos alterar o método sacar e iremos tambem acrescentar um Singleton, para que nossa classe seja instanciada uma única vez e com visibilidade e acessibilidade global dessa instância. Segue o código abaixo:

```
public class BancoServiceImpl implements BancoService {

	private LocalDate data = LocalDate.now();
	private LocalTime hora = LocalTime.now();

	private static BancoServiceImpl instance;

	private BancoServiceImpl(){
	}

	public static BancoServiceImpl getInstance(){
		if(instance == null){
			instance = new BancoServiceImpl();
		}
		return instance;
	}

	public void sacar(double valor){
		System.out.println("Banco Triway: Sacado R$ "+ valor+" no dia"
		+data.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"))+" às "
		+hora.format(DateTimeFormatter.ofPattern("HH:mm")));
	}
}
```

SINGLETON

3. Certo, agora para podermos injetar essa classe **BancoService** em qualquer outra classe do projeto, devemos criar um produtor dela. Crie agora a classe **BancoFactory** e siga o exemplo do código a seguir.

```
public class BancoFactory {

	@Produces
	public BancoService createBanco(){
		return new BancoServiceImpl();
```

```
        }
    }
```

4. Agora vamos começar o teste. Crie o bean **BancoTeste** como mostrado abaixo.

```
public class BancoTeste {

        private double valor;

        @Inject
        private BancoService bancoService;

        public void testCDI(){
              bancoService.sacar(valor);
        }
        //getter e setter omitidos
}
```

5. Desenvolva uma página xhtml simples para testar o seu produtor.

```
<h:body>

      <h:form>
            <p:outputLabel value="ESCOLHA A QUANTIDADE A SER SACADA" />
            <br/>
            <p:inputText value="#{bancoTeste.valor}" />
            <p:commandButton value="Sacar" action="#{bancoTeste.testeCDI()}" />
      </h:form>
</h:body>
```

Perceba que ao fazer o teste, é apontado um problema de ambiguidade. Para resolver esse problema, vamos criar uma anotação **@BancoProducer** para diferenciar a classe implementada do produtor. Segue de exemplo o código abaixo:

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.TYPE})
public @interface BancoProducer {

}
```

Iremos colocar essa anotação na classe **BancoFactory**, logo após a anotação @Produces.

```
@Produces @BancoProducer
public BancoService createBanco(){
```

Faça o teste e veja o resultado. Tente acrescentar novas informações ou novos bancos através de qualificadores, como passado no exercicio 3.

Exercício 5 – Alternatives

1. Em nosso projeto, podemos acrescentar implementações alternativas de um mesmo método usando a tag **@Alternative.** Para fazermos isso, Crie uma classe **BancoServiceAlt** com as mesmas configurações da BancoServiceImpl, porém com algumas alterações, para que se diferencie da classe original.

```
public void sacar(double valor) {
```

```
        System.out.println("Banco Alternativo: Sacado R$ "+valor);
    }
```

2. Acrescente a tag **@Alternative** acima da classe **BancoServiceAlt**.

```
@Alternative
public class BancoServiceAlt implements BancoService {

        ...
}
```

3. Vá ao arquivo beans.xml do seu projeto e acrescente a tag <alternatives> junto com a classe que foi marcada pela anotação. Isso fará com que a classe anotada seja chamada na injeção.

```
<beans … >
        <alternatives>
                <class>cdi.BancoServiceAlt</class>
        </alternatives>
</beans>
```

4. Como esse modelo de injeção não depende do @Produces para funcionar, então é necessário comentar a classe **BancoFactory** para não haver conflitos. Faça os testes com e sem a tag <alternatives> no **beans.xml**.

Exercício 6 – Interceptadores e Logger

1. Vamos escrever uma aplicação simples que nos permite reservar bilhetes para um filme. Crie uma anotação @Log que funcionará como um InterceptorBinding.

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

        }
```

2. Criada a nova anotação, vamos criar um **LoggingInterceptor**.

```
@Interceptor
 @Log

public class LoggingInterceptor implements Serializable{

         @AroundInvoke
public Object logMethodEntry(InvocationContext ctx) throws Exception {
                System.out.println("Entrando método:"+
                ctx.getMethod().getName());

                return ctx.proceed();
        }
}
```

3. Crie a classe **BookShow** onde ficará os métodos a serem utilizados nesse exercício.

```
@Log

@Named
@SessionScoped
public class BookShow implements Serializable {

	private static final long serialVersionUID = 6350400892234496909L;

	public List<String> getBookList() {
		List<String> booksAvailable = new ArrayList<String>();
		booksAvailable.add("O Livro Proibido");
		booksAvailable.add("Biblia do Programador");
		return booksAvailable;
	}

	public Integer getDiscountedPrice(int bookPrice) {
		return bookPrice - 50;
	}
}
```

4. Para esse interceptador funcionar, deve ser especificado no beans.xml assim como mostrado abaixo.

```
<beans … >

	<interceptors>
		<class>cdi2.LoggingInterceptor</class>
	</interceptors>

</beans>
```

5. Agora faça uma aplicação web para testar o nosso **LoggingInterceptor.**

```
<h:body>

	<h:form>
		<h:panelGrid columns="2" >
			<h:outputLabel value="Teste CDI" />
			<h:commandButton action="#{bookShow.getBooksList}"
			value="Interceptar" />
		</h:panelGrid>
	</h:form>
</h:body>
```

Faça o teste e veja os resultados.

```
- Entrando método: getBookList
```

6. Vamos agora mostrar essa mesma mensagem com o Logger. Volte na classe LoggingInterceptor, comente o println e faça a seguinte alteração:

```
private static final Logger logger =
	Logger.getLogger (LoggingInterceptor.class.getName());



@AroundInvoke
public Object logMethodEntry(InvocationContext ctx) throws Exception {
	logger.info("Entrando método:"+ctx.getMethod().getName()+" | "
		+logger.getName());
	… }
```

Faça os testes e veja o resultado.

```
ago 08, 2016 2:27:28 PM cdi.LoggingInterceptor logMethodEntry
INFO: Entrando método: getBookList | cdi.LoggingInterceptor
```