


Inversão de Controle

O Framework vai entregar a dependência.
Sua classe não precisa busca-la.

```
public class ClienteDaoImpl extends DaoImpl<Cliente> {  
    private static final long serialVersionUID = -1081947125480849486L;  
  
    private EntityManager manager;  
  
    public ClienteDaoImpl(EntityManager manager) {  
        this.manager = manager;  
    }  
}
```

EntityManager é passado
como parâmetro no construtor



Utilizar a injeção de dependências!

Injeção de Dependências

Anotação *@Inject* indica qual dependência está sendo injetada

Injeção de um
EntityManager

```
public class ClienteDaoImpl extends DaoImpl<Cliente> {  
  
    private static final long serialVersionUID = -1081947125480849486L;  
  
    @Inject  
    private EntityManager manager;  
}
```

A classe pode receber qualquer tipo de dependência.

Injeção de Dependências

Injeção através de campos (atributos)

```
public class exemploUsoCdi {  
  
    @Inject  
    @Exemplo01  
    private ExemploInterface service;  
  
}
```

@Exemplo1 seria a implementação da interface que estamos injetando (poderia ser *@Exemplo2*)

Injeção de Dependências

Injeção através de construtores

```
public class exemploUsoCdi {  
  
    private ExemploInterface service;  
  
    @Inject  
    public exemploUsoCdi (@Exemplo01 ExemploInterface service)  
    {  
        this.service = service;  
    }  
  
}
```

@Exemplo1 seria a implementação da interface que estamos injetando (poderia ser @Exemplo2)

Injeção de Dependências

Injeção através de *setters*

```
public class exemploUsoCdi {  
  
    private ExemploInterface service;  
  
    @Inject  
    public void setService(@Exemplo01 ExemploInterface service) {  
        this.service = service;  
    }  
  
}
```

@Exemplo1 seria a implementação da interface que estamos injetando (poderia ser @Exemplo2)

Beans

```
public class Carro {  
    private String modelo;  
    private String chassi;  
  
    public Carro() { }
```

Classe não estática

Construtor vazio ou anotado com *@Inject*


Classe não declarada como um bean EJB

Classe não anotada com um componente EJB

Managed Beans

@Named

```
@Named  
@RequestScoped  
public class LoginBean implements Serializable {
```

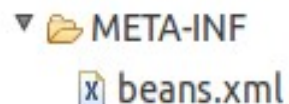


A anotação *@Named* integra o CDI com JSF, dando um nome ao bean.

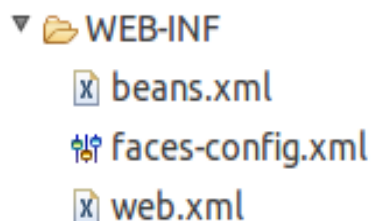
```
<p:commandButton action="#{loginBean.realizarLogin()}"
```


Configuração

Criar um arquivo *beans.xml* na pasta
META-INF



Ou na WEB-INF caso seja um projeto web

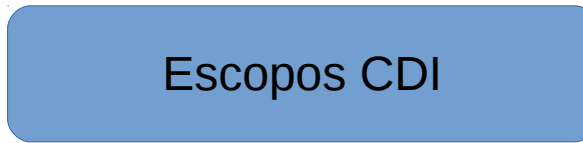
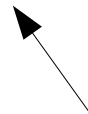


Conteúdo do arquivo

```
<?xml version="1.0"?>
<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://jboss.org/schema/cdi/beans_1_1.xsd">
</beans>
```

RequestScoped (@RequestScoped)

SessionScoped (@SessionScoped)



ApplicationScoped (@ApplicationScoped)

Dependent (@dependent)



Convesation Scoped (@ConversationScoped)

Escopos CDI

Funcionam como os escopos do JSF

RequestScoped (@RequestScoped)

Começa na requisição e termina quando o servidor devolve a resposta.

SessionScoped (@SessionScoped)

Começa com a primeira requisição feita e termina quando a aplicação encerra a sessão ou por inatividade.

ApplicationScoped (@ApplicationScoped)

Dura enquanto a aplicação estiver ativa.

Agora temos *Conversation Scoped*

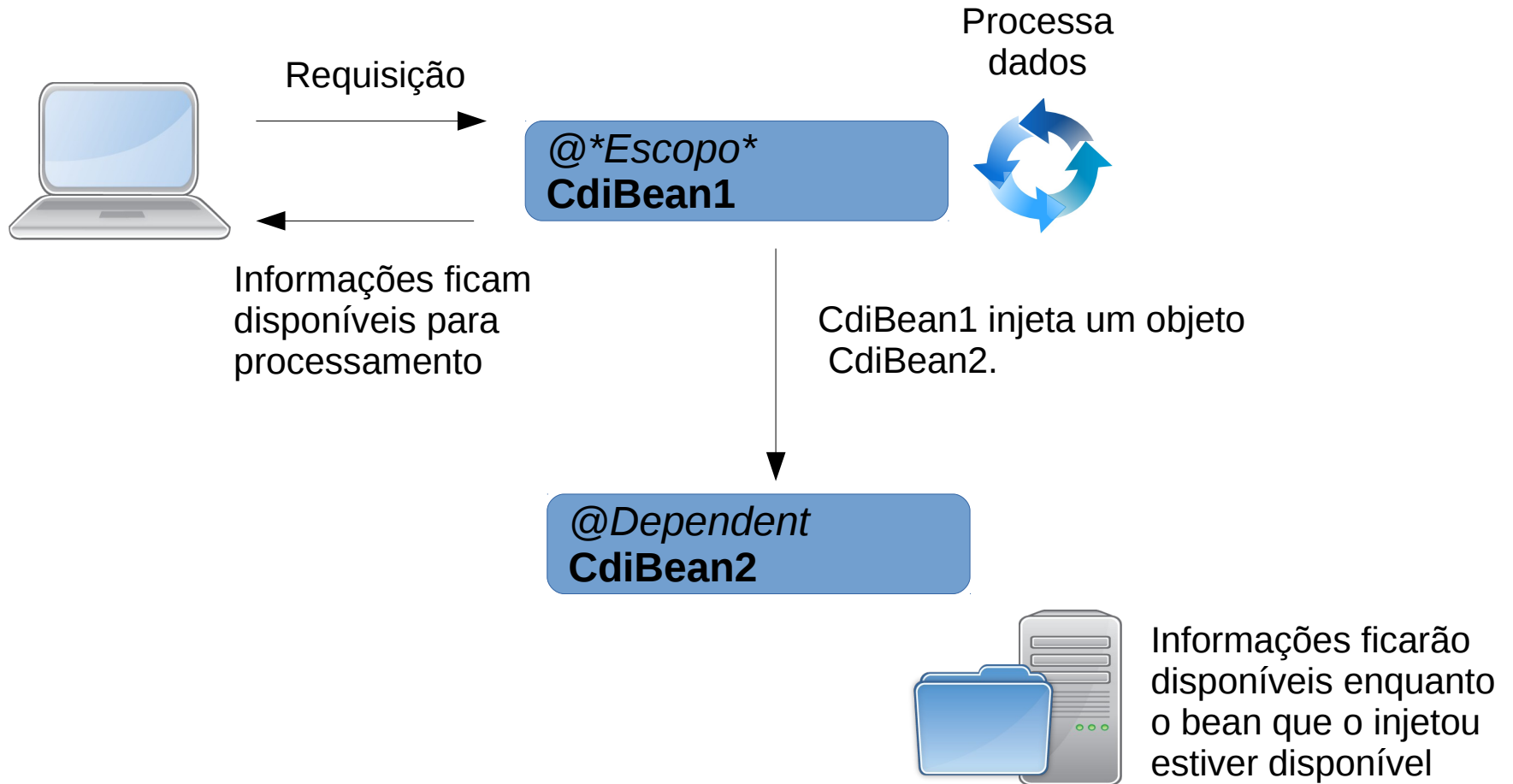
Escopo de Conversação

@ConversationScoped mantém as informações durante a comunicação entre cliente e servidor



Escopo Dependente

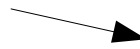
@Dependent é o escopo padrão. Quando injetamos um objeto em um bean, esse objeto tem o escopo do bean.



Produtores e Escopos

Anotação *@produces*

Quando será
produzido?



```
@Produces
@RequestScoped
public EntityManager createEntityManager() {

    return factory.createEntityManager();
}
```

Decisão do que
será produzido
com estruturas
de decisão

Métodos produtores
são uma forma fácil
de integrar objetos
que não são beans
ao ambiente CDI

Fechando o EntityManager

@Disposes

```
public void closeEntityManager(@Disposes EntityManager manager) {

    manager.close();
}
```

Ambiguidade

Utilização de qualifiers

Supondo que estajamos injetando uma Connection, mas possuímos um banco MySQL e um PostgreSQL

```
public class ClienteDao implements Dao<Cliente> , Serializable{  
  
    /** Atributo serialVersionUID. */  
    private static final long serialVersionUID = -2128800246766559094L;  
  
    @Inject  
    Connection conexao;
```



Qual conexão?

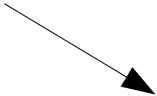
Qualifiers

```
import static java.lang.annotation.ElementType.FIELD;  
import static java.lang.annotation.ElementType.METHOD;  
import static java.lang.annotation.ElementType.PARAMETER;  
import static java.lang.annotation.ElementType.TYPE;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;
```

```
import javax.inject.Qualifier;
```


Devemos criar
um qualifier para
que a CDI
diferencie o tipo
de conexão.



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


Qualifiers

Padrão caso a
injeção não seja
especificada




```
@Default
@Produces
public static Connection getConexao() throws SQLException {

    try{
        Class.forName("org.postgresql.Driver");
        return DriverManager.getConnection(url,usuario,senha);

    }catch(ClassNotFoundException e){
        throw new SQLException(e.getMessage());
    }
}
```

Qualifier criado
anteriormente



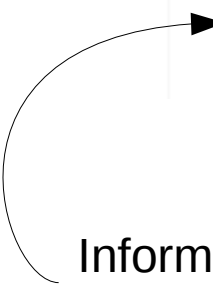
```
@ConexaoMySQL
@Produces
public static Connection getConexaoMySQL() throws SQLException {

    try{
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection(url,usuario,senha);

    }catch(ClassNotFoundException e){
        throw new SQLException(e.getMessage());
    }
}
```

Qualifier

```
public class ClienteDao implements Dao<Cliente> , Serializable{  
  
    /** Atributo serialVersionUID. */  
    private static final long serialVersionUID = -2128800246766559094L;  
  
    @Inject  
    @ConexaoMYSQL  
    Connection conexao;
```



Informamos qual conexão deverá ser usada.

Qualifier com ENUMS

```
@Retention(RUNTIME)
@Qualifier
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Conexao {
    EnumConexao value() default EnumConexao.POSTGRES;
}

public enum EnumConexao {

    MYSQL, POSTGRES
}
```

```
public class ClienteDao implements Dao<Cliente> , Serializable{

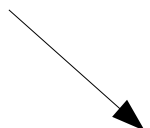
    /** Atributo serialVersionUID. */
    private static final long serialVersionUID = -2128800246766559094L;

    @Inject
    @Conexao(EnumConexao.POSTGRES)
    Connection conexao;
}
```

Alternatives

Anotação

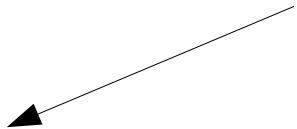
@Alternative permite especificar qual bean utilizar, basta indicar no *beans.xml*



```
public class ImplementacaoPadrao implements ExemploInterface {  
    //Implementação padrão  
}  
  
@Alternative  
public class ImplementacaoAlternativa implements ExemploInterface {  
    //implementação alternativa  
}
```

Alternative usando Esteriótipo

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



```
@ImplAlternativa  
public class ImplementacaoAlternativa implements ExemploInterface {  
  
    //implementação alternativa  
}
```

Declaração no
beans.xml

```
<alternatives>  
    <stereotype>exemploCdi.ImplAlternativa</stereotype>  
</alternatives>
```

Interceptadores

Anotação *@InterceptorBinding*

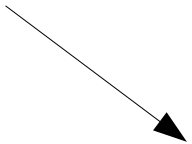
Criamos uma anotação com ***@InterceptorBinding*** para vincularmos um interceptador

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.interceptor.InterceptorBinding;

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

}
```



Interceptadores

Indica que essa classe será a interceptadora

```
@Interceptor
@Transactional
public class TransactionInterceptor implements Serializable {
```

```
    private static final long serialVersionUID = 1L;
```

```
    @Inject
    private EntityManager manager;
```

A anotação **@AroundInvoke** indica para o sistema qual método realizará a interceptação dos métodos de negócio

```
@AroundInvoke
public Object invoke(InvocationContext context) throws Exception {
```

```
    EntityTransaction trx = manager.getTransaction();
```

```
    boolean criador = false;
```

```
    try {
```

```
        if (!trx.isActive()) {
```

```
            trx.begin();
```

```
            trx.rollback();
```

```
            trx.begin();
```

```
            criador = true;
```

```
        }
```

```
        return context.proceed();
```

```
    } catch (Exception e) {
```

```
        if (trx != null && criador) {
```

```
            trx.rollback();
```

```
        }
```

```
        throw e;
```

```
    } finally {
```

```
        if (trx != null && trx.isActive() && criador) {
```

```
            trx.commit();
```

```
        }
```

```
    }
```

```
}
```

```
}
```

Interceptadores

```
@Transactional  
public class ClienteDaoImpl extends DaoImpl<Cliente> {
```

@Transactional indica que todos os métodos dessa classe serão interceptados. Podemos também utilizar a anotação somente em um método específico.

Declaração no *beans.xml*

```
<interceptors>  
  <class>threeway.projeto.service.DaoJPA.transactions.TransactionInterceptor</class>  
</interceptors>
```


Eventos

Utilizamos a anotação *@Observes* para monitorar quando algum evento for disparado

Injeção de um evento do tipo Carro

```
public class FabricaCarro {  
    @Inject  
    private Event<Carro> eventoCarro;  
  
    public void montarCarro (Carro carro) {  
  
        System.out.println("Montando carro...");  
        eventoCarro.fire(carro);  
    }  
}
```

Dispara o evento

```
public class NotificarClientes {  
  
    public void notificarClientes(@Observes Carro carro) {  
  
        //Envia uma notificação aos clientes da montadora  
        //dizendo que o carro está sendo montado  
    }  
}
```

Classe com o método que “observa” o evento (*@Observes*) e realiza alguma ação após ele ser disparado.

Decorator

Funcionamento parecido com interceptadores

```
public class FabricaCarro {  
    public void montarCarro () {  
        System.out.println("Montando carro...");  
    }  
}  
  
@Decorator  
public class FabricaCarroDecorator {  
    @Inject  
    @Delegate  
    private FabricaCarro fabricaCarro;  
  
    public void montarCarro() {  
        //Envia uma notificação aos clientes da montadora  
        //dizendo que o carro está sendo montado  
  
        fabricaCarro.montarCarro();  
    }  
}
```

Indica o bean a ser interceptado



Declaração do Decorator no *beans.xml*

```
<decorators>  
    <class>exemplosCdi.FabricaCarroDecorator</class>  
</decorators>
```

CDI e JPA/EJB

Os beans do EJB são:

- Transacionais
- Remotos ou locais
- Podem passivar Stateful beans, liberando recursos
- Podem fazer o uso de *Timer*
- Podem ser assíncronos

Um bean anotado com **@Stateless** não precisa lidar com transações, o servidor irá cuidar disso.

Caso queira utilizar JPA em um container como Tomcat, as transações devem ser feitas manualmente.

Beans EJB são beans CDI, portanto tem os mesmos benefícios, mas dizer o contrário não é válido.

CDI e JSF

Anotações de escopo do CDI
javax.enterprise.context

```
import javax.enterprise.context.RequestScoped;  
  
@Named  
@RequestScoped  
public class LoginBean implements Serializable {
```

Anotações de escopo do JSF
javax.faces.bean

```
import javax.faces.bean.ViewScoped;  
  
@ManagedBean  
@ViewScoped  
public class HelloWorldBean implements Serializable {
```