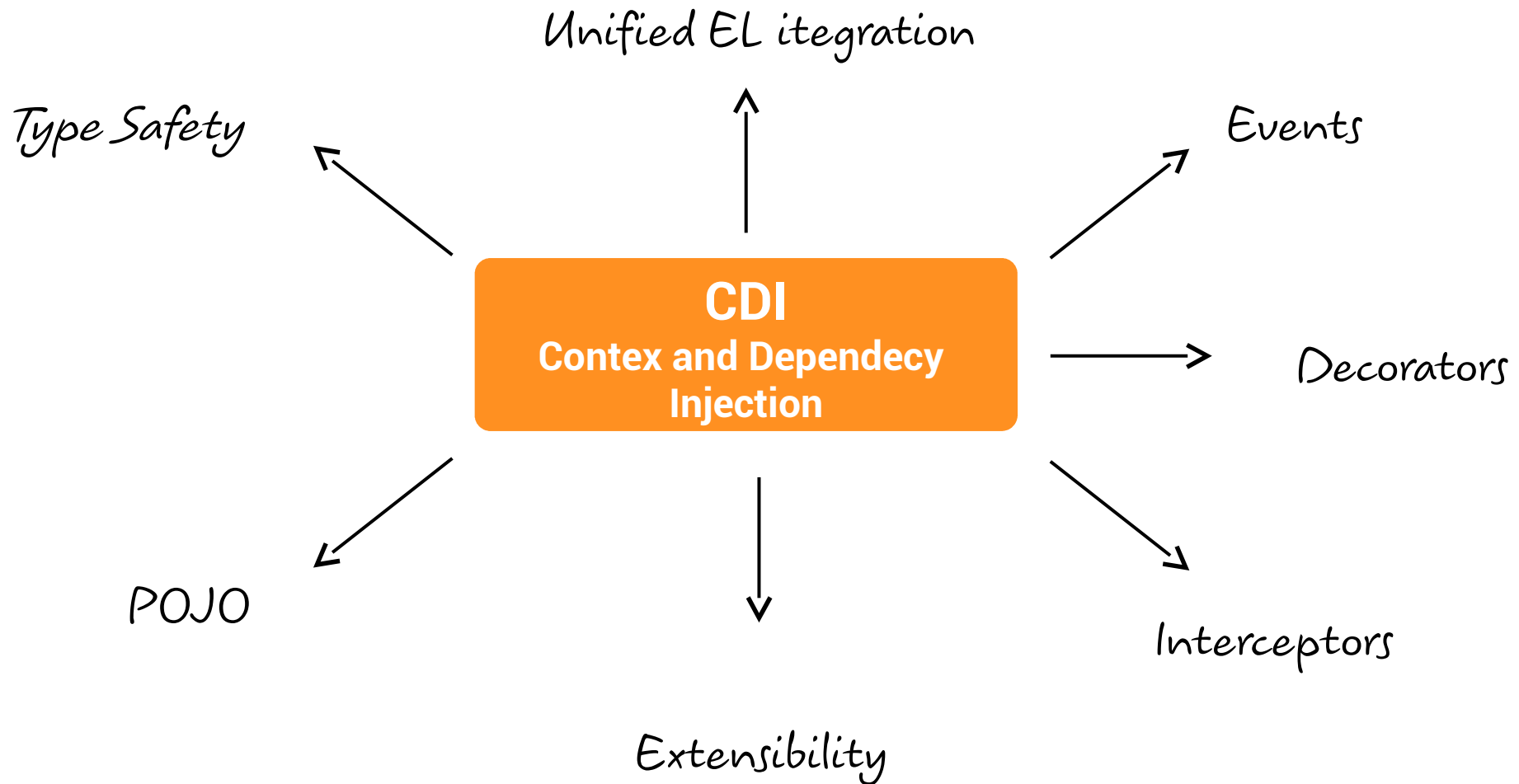


# CDI Context and Dependency Injection

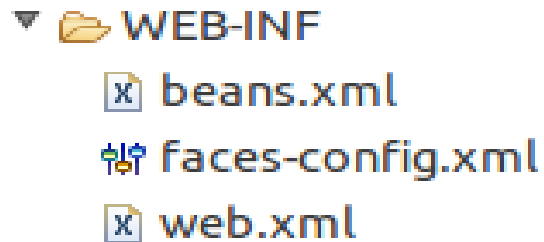


# Configuração

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



Ou na WEB-INF caso seja um projeto web



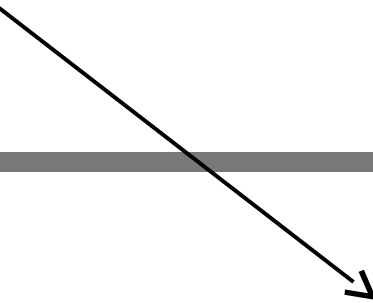
Conteúdo do arquivo

```
<?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>
```

# 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!

# 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

# 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.

## Ponto de Injeção – No Campo

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    protected Connection conexao;  
}
```

← Injeta um objeto `java.sql.Connection`, no campo

Injeta uma implementação da interface `Dao<T>`, no campo →

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

## Ponto de Injeção – No Construtor

Injeta uma implementação da interface `Dao<T>`, no construtor

```
public class MBLivro {  
    private Dao<?> dao;  
  
    @Inject  
    public MBLivro(Dao<?> dao) {  
        super();  
        this.dao = dao;  
    }  
}
```

```
public class AbstractDao<T> implements Dao<T> {  
    protected Connection conexao;  
  
    @Inject  
    public AbstractDao(Connection conexao) {  
        this.conexao = conexao;  
    }  
}
```

Injeta um objeto `java.sql.Connection`, no construtor

\* CUIDADO: você pode ter apenas um construtor com parâmetros injetados pelo CDI

## Ponto de Injeção – No Método de Configuração

```
public class AbstractDao<T> implements Dao<T> {  
    protected Connection conexao;  
  
    @Inject  
    public void setConexao(Connection conexao){  
        this.conexao = conexao;  
    }  
}
```

Injeta um objeto `java.sql.Connection`, no setter

Injeta uma implementação da interface `Dao<T>`, no setter

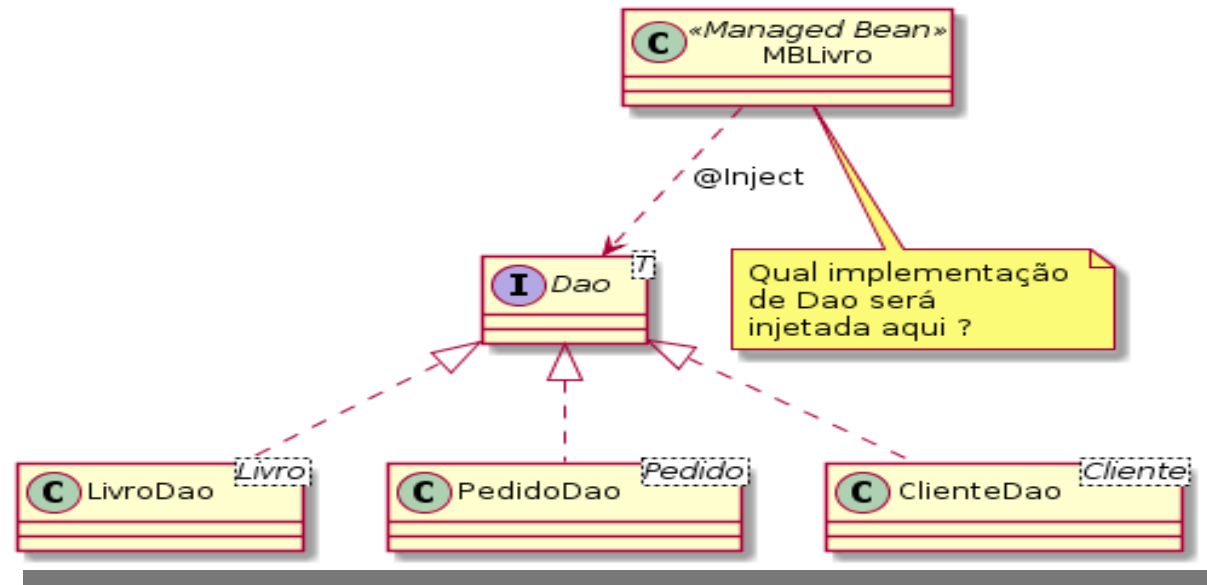
```
public class MBLivro {  
    private Dao<?> dao;  
  
    @Inject  
    public void setDao(Dao<?> dao) {  
        this.dao = dao;  
    }  
}
```



## Qualificadores – Ambiguidade de objetos

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

Várias implementações da mesma interface Dao, qual será injetada?



# Qualificadores - Desambiguidade

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

CDI precisa saber qual implementação da interface Dao deve ser usada, senão gerará uma mensagem de erro!

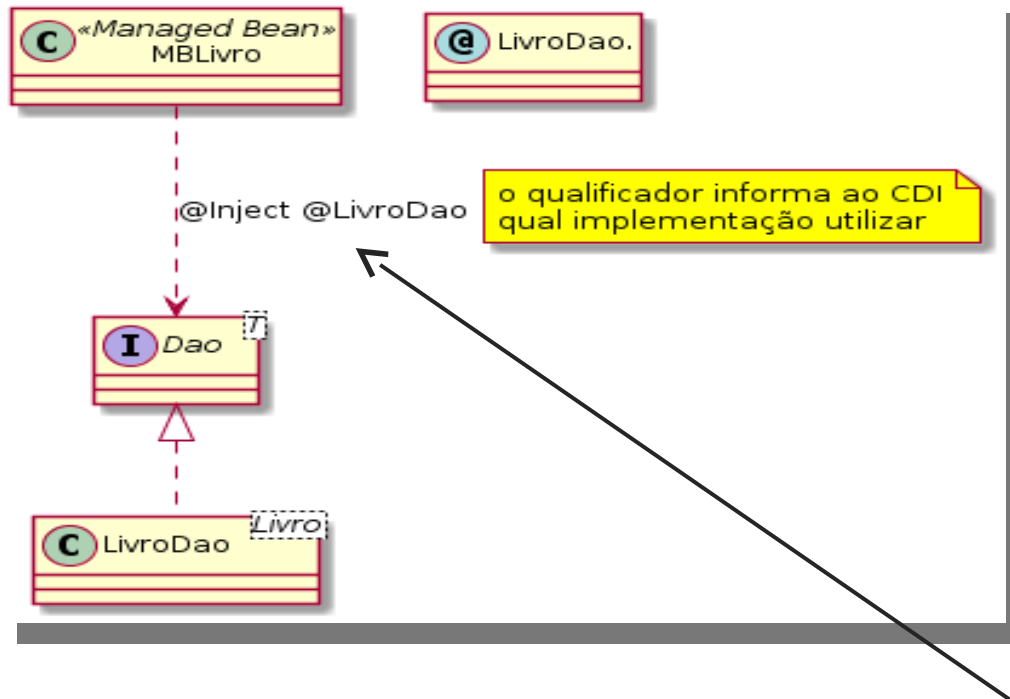
no eclipse Jboss tools

Configure Problem Severity for preference 'Unsatisfied or ambiguous dependencies f...'

com implementação CDI  
JBoss Weld

WELD-001409: Ambiguous dependencies for type Dao<Object> with qualifiers @Default at injection point [BackedAnnotatedField] @Inject private workshop.mbean.MBLivro.dao

# Qualificadores – Criando Anotações Qualificadoras



```
public class MBLivro {
```

```
    @Inject  
    @LivroDao  
    private Dao<?> dao;
```

```
import static java.lang.annotation.ElementType.*;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
import javax.inject.Qualifier;  
  
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ FIELD, METHOD, PARAMETER, TYPE })  
public @interface LivroDao {  
  
}
```

# Qualificadores – Usando Qualificadores

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

Anotar implementação

```
@annotation.PedidoDao
public class PedidoDao
    extends AbstractDao<Pedido>{

    @Inject
    private Connection conexao;
```

Qualificar ponto de injeção

```
public class MBExemplo {
    @Inject
    @PedidoDao
    private Dao<?> pDao;

    private Dao<?> lDao;

    @Inject
    public MBExemplo(@LivroDao Dao<?> dao) {
        this.lDao = dao;
    }
}
```

Outra implementação, outro qualificador @LivroDao

# Qualifier com ENUMS

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

```
public enum EnumConexao {

    MYSQL, POSTGRES

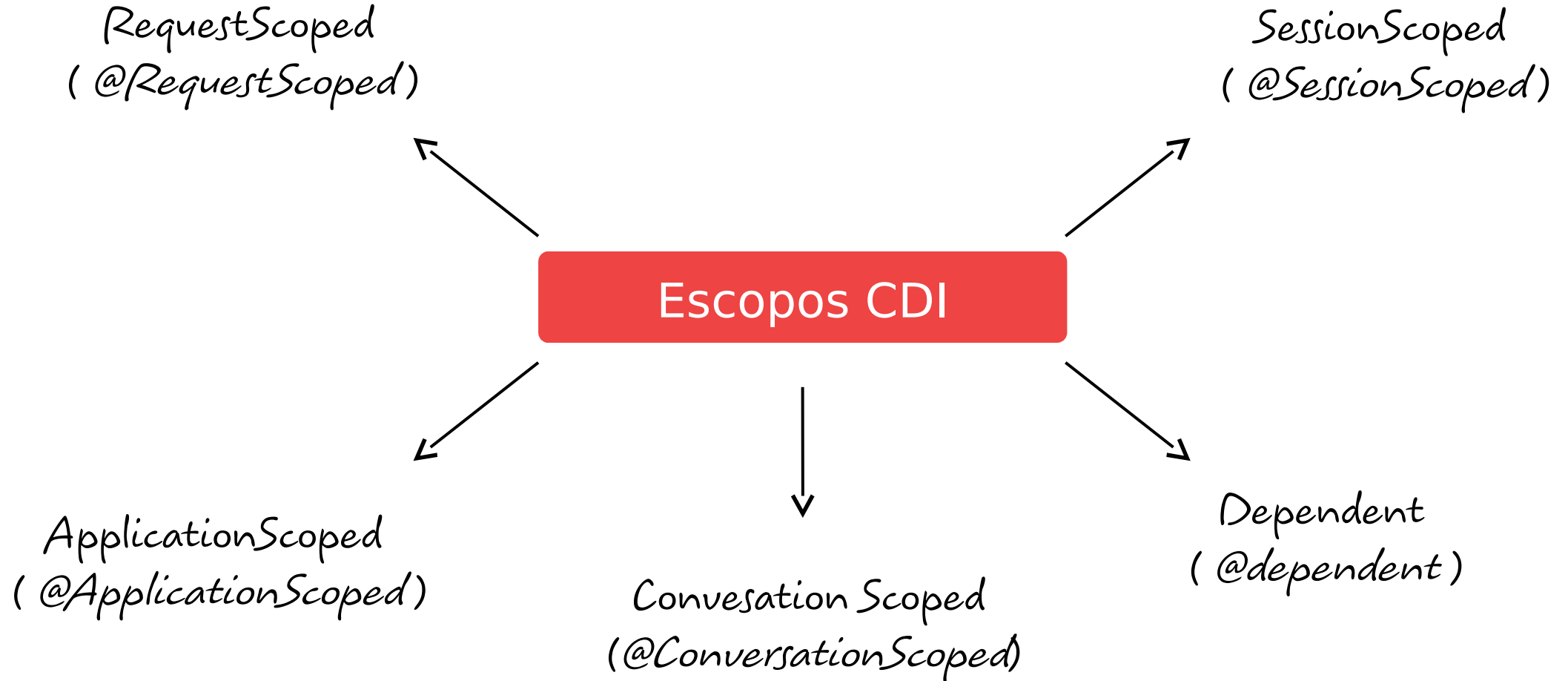
}
```

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

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

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

# Escopos CDI



# 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.

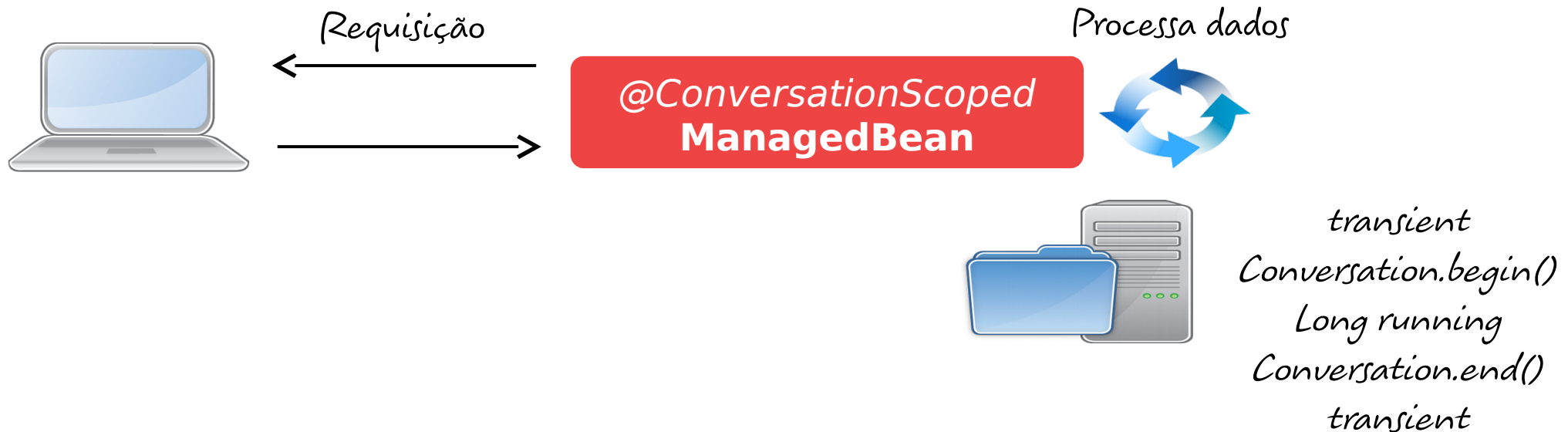
## **ApplicationScoped ( @ApplicationScoped )**

Dura enquanto a aplicação estiver ativa.

*Agora temos Conversation Scoped*

# Escopo de Conversação

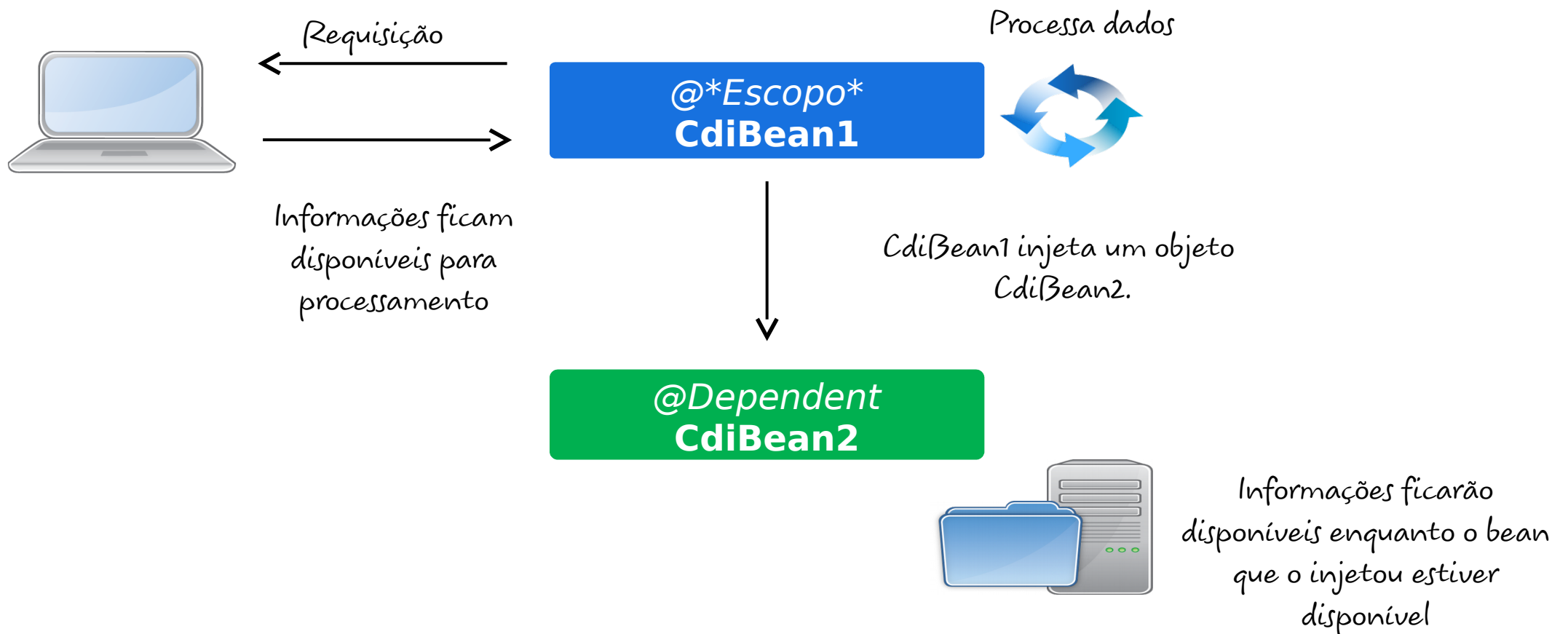
*@ConversationScoped mantém as informações durante a comunicação entre cliente e servidor*





# Escopo Dependente

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



# Produtores e Escopo

Quando será  
produzido?

```
@Produces  
@RequestScoped  
public EntityManager createEntityManager() {  
    return factory.createEntityManager();  
}
```

O tipo de retorno  
define o que será  
produzido

Fechando o EntityManager  
*@Disposes*

```
public void closeEntityManager(@Disposes EntityManager manager) {  
    manager.close();  
}
```

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

# 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 Estereótipo

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

```
@ImplAlternativa
public class ImplementacaoAlternativa implements ExemploInterface {

    //implementação alternativa
}
```

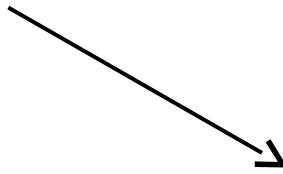
Declaração no beans.xml

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

```
<alternatives>
  <class>exemplosCdi.ImplemetacaoAlternativa</class>
</alternatives>
```

# Interceptadores Anotação `@InterceptorBinding`

Criamos uma anotação com  
**`@InterceptorBinding`**  
para vincularmos um  
interceptor



```
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;

    @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();
            }
        }
    }
}
```

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

# 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>
```

# Injeção a partir métodos Produtores

```
import java.sql.Connection;

public class FabricaConexao {
    static String url = "jdbc:postgresql://localhost:5432/Livraria";
    static String usuario = "postgres";
    static String senha = "postgres";

    @Produces
    public static Connection getConexao() throws SQLException{
        return DriverManager.getConnection(url,usuario,senha);
    }

    public static void closeConnection(@Disposes Connection conn)
        throws SQLException{
        System.out.println("FabricaConexao.closeConnection()");
        conn.close();
    }
}
```

Objetos que não  
são managed  
bean CDI podem  
ser injetados com  
@Producers

← @Disposes  
fechando o Connection

```
public class AbstractDao<T> implements Dao<T> {
    protected Connection conexao;

    @Inject
    public AbstractDao(Connection conexao) {
        this.conexao = conexao;
    }
}
```



# Injeção de EJB a partir de Produtores

Beans EJB não são managed bean CDI, mas podem ser injetados com @Producers

```
public class FabricaConexao {  
    @Produces  
    @PersistenceContext  
    private EntityManager em;  
  
    public void closeConnection(@Disposes EntityManager em)  
        throws SQLException{  
        em.close();  
    }  
}
```

@Disposes  
fechando o EntityManager

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    protected EntityManager em;  
}
```

# Desambiguação de Produtores

@Producers ambíguos devem ser qualificados

```
public class FabricaConexao {  
    @Produces  
    @PersistenceContext(unitName="pu-prod")  
    private EntityManager prodEm;  
  
    @Produces  
    @PersistenceContext(unitName="pu-test")  
    private EntityManager testEm;  
}
```

Crie @Qualifier's para diferenciar

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

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

# Desambiguação de Produtores

@Producers qualificados

```
public class FabricaConexao {  
    @Produces  
    @ProdDatabase  
    @PersistenceContext(unitName="pu-prod")  
    private EntityManager prodEm;  
  
    @Produces  
    @TestDatabase  
    @PersistenceContext(unitName="pu-test")  
    private EntityManager testEm;  
}
```

Use qualificador  
@Qualifier no ponto de  
injeção para  
diferenciar o produtor

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    @ProdDatabase  
    protected EntityManager em;  
}
```

## Ponto de Injeção – no campo

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    protected Connection conexao;  
}
```

Injeta um objeto  
java.sql.Connection, no campo

Injeta uma implementação da  
interface Dao<T>, no campo

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

## Ponto de Injeção – no construtor

Injeta uma implementação da interface `Dao<T>`, no construtor

```
public class MBLivro {  
    private Dao<?> dao;  
  
    @Inject  
    public MBLivro(Dao<?> dao) {  
        super();  
        this.dao = dao;  
    }  
}
```

```
public class AbstractDao<T> implements Dao<T> {  
    protected Connection conexao;  
  
    @Inject  
    public AbstractDao(Connection conexao) {  
        this.conexao = conexao;  
    }  
}
```

Injeta um objeto `java.sql.Connection`, no construtor

\* CUIDADO: você pode ter um construtor com parâmetros injetados pelo CDI

## Ponto de Injeção – no método de configuração

```
public class AbstractDao<T> implements Dao<T> {  
    protected Connection conexao;  
  
    @Inject  
    public void setConexao(Connection conexao){  
        this.conexao = conexao;  
    }  
}
```

Injeta um objeto  
java.sql.Connection, no setter

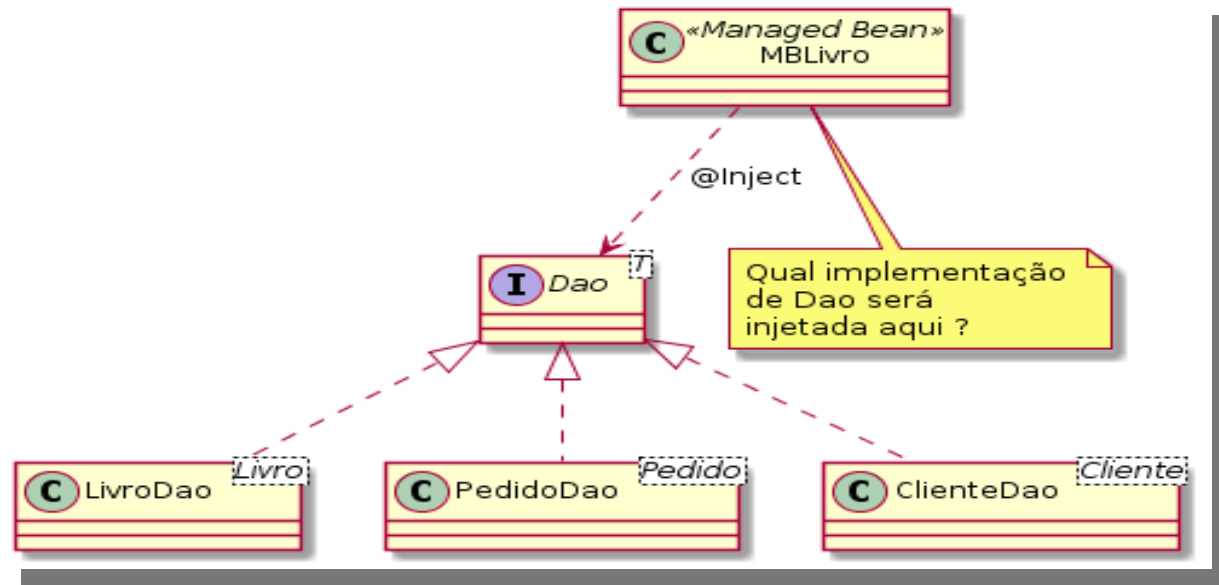
Injeta uma implementação da  
interface Dao<T>, no setter

```
public class MBLivro {  
    private Dao<?> dao;  
  
    @Inject  
    public void setDao(Dao<?> dao) {  
        this.dao = dao;  
    }  
}
```

# Qualificadores – ambiguidade de objetos

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

Várias implementações da mesma interface Dao, qual será injetada?



# Qualificadores – Desambiguação

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;  
}
```

CDI precisa saber qual implementação da interface Dao deve ser usada, senão gerará uma mensagem de erro!

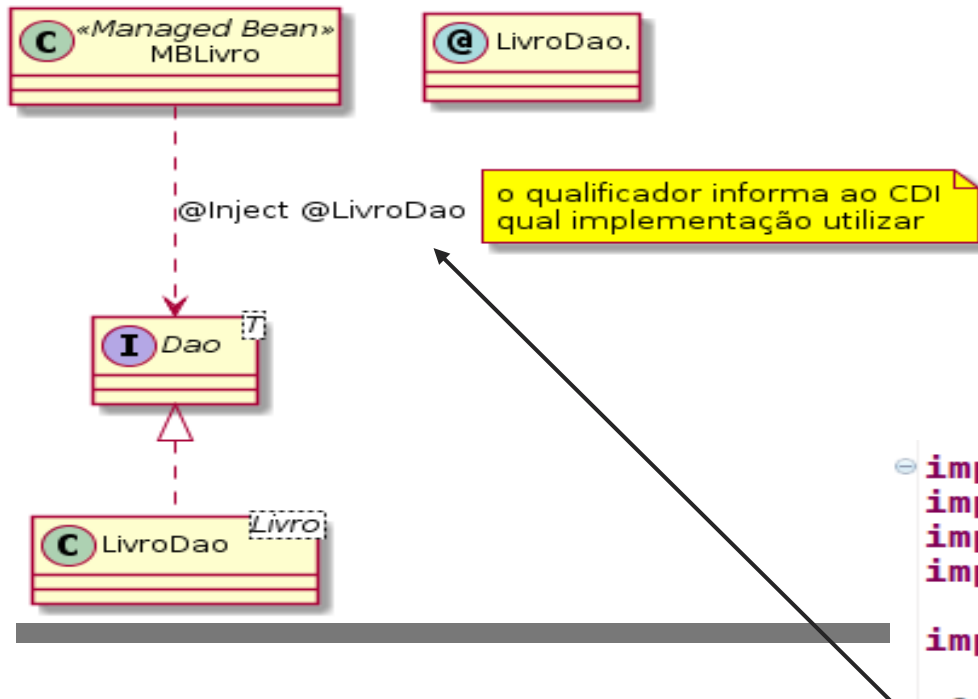
no eclipse Jboss tools

com implementação  
CDI JBoss Weld

WELD-001409: Ambiguous dependencies for type Dao<Object> with qualifiers @Default at injection point [BackedAnnotatedField] @Inject private workshop.mbean.MBLivro.dao



# Qualificadores – criando anotações qualificadoras



```
public class MBLivro {  
    @Inject  
    @LivroDao  
    private Dao<?> dao;  
}
```

```
import static java.lang.annotation.ElementType.*;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
import javax.inject.Qualifier;  
  
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ FIELD, METHOD, PARAMETER, TYPE })  
public @interface LivroDao {  
  
}
```

# Qualificadores – usando qualificadores

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

Anotar implementação

```
@annotation.PedidoDao
public class PedidoDao
    extends AbstractDao<Pedido>{

    @Inject
    private Connection conexao;
}
```

Qualificar ponto de injeção

```
public class MBExemplo {
    @Inject
    @PedidoDao
    private Dao<?> pDao;

    private Dao<?> lDao;

    @Inject
    public MBExemplo(@LivroDao Dao<?> dao) {
        this.lDao = dao;
    }
}
```

Outra implementação, outro qualificador  
@LivroDao

# Injeção a partir métodos Produtores

Objetos que não são managed bean CDI podem ser injetados com @Producers

```
import java.sql.Connection;

public class FabricaConexao {
    static String url = "jdbc:postgresql://localhost:5432/Livraria";
    static String usuario = "postgres";
    static String senha = "postgres";

    @Produces
    public static Connection getConexao() throws SQLException{
        return DriverManager.getConnection(url,usuario,senha);
    }

    public static void closeConnection(@Disposes Connection conn)
        throws SQLException{
        System.out.println("FabricaConexao.closeConnection()");
        conn.close();
    }
}
```

↑  
@Disposes fechando o Connection

```
public class AbstractDao<T> implements Dao<T> {
    protected Connection conexao;

    @Inject
    public AbstractDao(Connection conexao) {
        this.conexao = conexao;
    }
}
```

# Injeção de EJB a partir de Produtores

Beans EJB não são managed bean CDI, mas podem ser injetados com @Producers

```
public class FabricaConexao {  
    @Produces  
    @PersistenceContext  
    private EntityManager em;  
  
    public void closeConnection(@Disposes EntityManager em)  
        throws SQLException{  
        em.close();  
    }  
}
```

@Disposes  
fechando o EntityManager

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    protected EntityManager em;
```

# Desambiguação de Produtores

@Producers ambíguos  
devem ser qualificados

```
public class FabricaConexao {  
    @Produces  
    @PersistenceContext(unitName="pu-prod")  
    private EntityManager prodEm;  
  
    @Produces  
    @PersistenceContext(unitName="pu-test")  
    private EntityManager testEm;  
}
```

Crie @Qualifier's  
para diferenciar

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

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

# Desambiguação de Produtores

@Producers qualificados

```
public class FabricaConexao {  
    @Produces  
    @ProdDatabase  
    @PersistenceContext(unitName="pu-prod")  
    private EntityManager prodEm;  
  
    @Produces  
    @TestDatabase  
    @PersistenceContext(unitName="pu-test")  
    private EntityManager testEm;  
}
```

Use qualificador  
@Qualifier no ponto de  
injeção para  
diferenciar o produtor

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    @ProdDatabase  
    protected EntityManager em;  
}
```

## Alternatives – Usando implementação alternativa

```
public class FabricaConexao {  
    @Produces  
    @Default  
    @PersistenceContext(unitName = "pu-test")  
    private EntityManager testEm;  
  
    @Produces  
    @Alternative  
    @DevDatabase  
    @PersistenceContext(unitName = "pu-dev")  
    private EntityManager devEm;  
  
    @Produces  
    @Alternative  
    @ProdDatabase  
    @PersistenceContext(unitName = "pu-prod")  
    private EntityManager prodEm;  
}
```

```
public class AbstractDao<T>  
    implements Dao<T> {  
    @Inject  
    protected EntityManager em;  
}
```

- 1ª Por padrão, crie um EntityManager testEm.
- 2ª Se a alternativa @DevDatabase estiver ativo, produza devEm.
- 3ª Se a alternativa @ProdDatabase estiver ativo, produza prodEm

# Alternatives – Usando implementação alternativa

```
public class FabricaConexao {  
    @Produces  
    @Default  
    @PersistenceContext(unitName = "pu-test")  
    private EntityManager testEm;  
  
    @Produces  
    @Alternative  
    @DevDatabase  
    @PersistenceContext(unitName = "pu-dev")  
    private EntityManager devEm;  
  
    @Produces  
    @Alternative  
    @ProdDatabase  
    @PersistenceContext(unitName = "pu-prod")  
    private EntityManager prodEm;  
}
```

```
public class AbstractDao<T>  
    implements Dao<T> {  
    @Inject  
    protected EntityManager em;  
}
```

beans.xml

```
<alternatives>  
    <stereotype>  
        DevDatabase  
    </stereotype>  
</alternatives>
```

Alternative ativo

- 1ª- Por padrão, crie um EntityManager testEm.
- 2ª- Se a alternativa @DevDatabase estiver ativo, produza devEm.
- 3ª- Se a alternativa @ProdDatabase estiver ativo, produza prodEm



# Alternatives – Seleção de alternativa por Estereótipo

```
@Alternative
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE,
          ElementType.METHOD,
          ElementType.FIELD })
@Documented
public @interface DevDatabase {
}
```

```
@Alternative
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE,
          ElementType.METHOD,
          ElementType.FIELD })
@Documented
public @interface ProdDatabase {
}
```

beans.xml

```
<alternatives>
  <stereotype>
    DevDatabase
  </stereotype>
</alternatives>
```

alternar implementação

beans.xml

```
<alternatives>
  <stereotype>
    ProdDatabase
  </stereotype>
</alternatives>
```

# Eventos

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

Injeção de um evento  
do tipo `Carro`

Dispara o evento

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

# Decorators – Decorando seu Bean

```
public class ReportService
implements IReport{

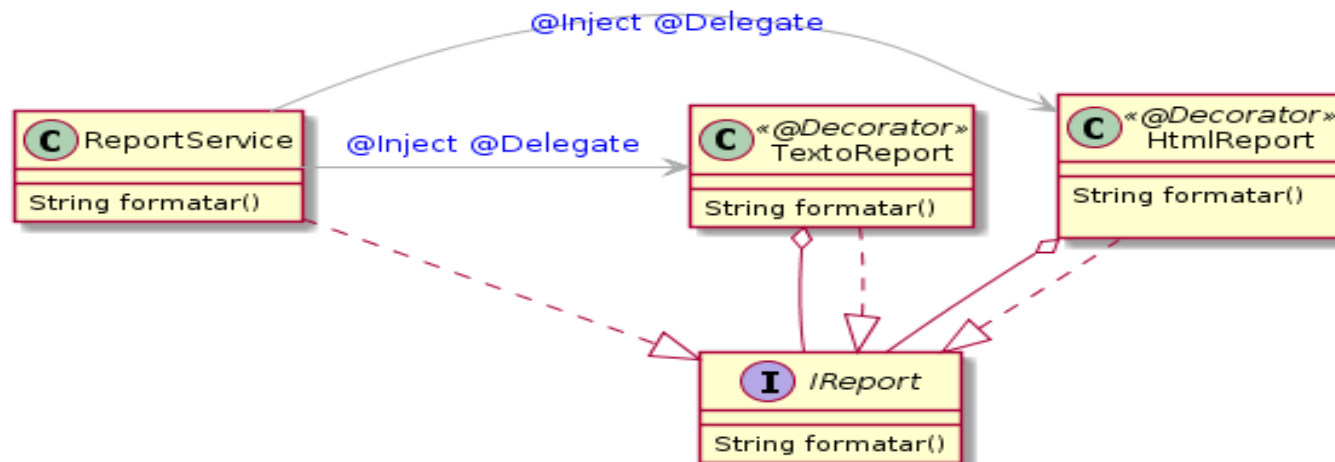
    @Override
    public String formatar() {
        return "Dados relatorio";
    }
}
```

Saída sem decorator

Dados relatorio

```
@Inject
private ReportService report;

public void showReport(){
    System.out.println(
        report.formatar()
    );
}
```



@Decorator, pattern Decorator: utilizado para adicionar novas responsabilidades a objetos dinamicamente

# Decorators – formatando html

```
@Decorator
public class HtmlReport implements IReport {
    @Inject
    @Delegate
    IReport report;

    @Override
    public String formatar() {
        return "<html><head>"
            + "<title>Relatorio HTML</title></head>"
            + "<body><p> Usando CDI Decorator </p><p>"
            + report.formatar() + "</p></body></html>";
    }
}
```

altera implementação usada

decora implementação existente

```
public class ReportService
implements IReport{

    @Override
    public String formatar() {
        return "Dados relatorio";
    }
}
```

```
<!-- beans.xml -->
<decorators>
    <class>org.cdi.decorators.HtmlReport</class>
</decorators>
```

injeta implementação  
"decorada"

```
@Inject
private ReportService report;

public void showReport(){
    System.out.println(
        report.formatar()
    );
}
```

saída decorada

```
<html>
<head>
    <title>Relatorio HTML</title>
</head>
<body>
    <p> Usando CDI Decorator </p>
    <p>Dados relatorio</p>
</body>
</html>
```

# Decorators – formatando texto

```
@Decorator
public class TextoReport implements IReport {
    @Inject @Delegate
    IReport report;

    @Override
    public String formatar() {
        return "Relatorio Texto \n\t\t"
            + "Usando CDI Decorator\n"
            + report.formatar();
    }
}
```

altera implementação usada

decora implementação existente

```
public class ReportService
implements IReport{

    @Override
    public String formatar() {
        return "Dados relatorio";
    }
}
```

```
<!-- beans.xml -->
<decorators>
    <class>org.cdi.decorators.TextoReport</class>
</decorators>
```

injeta implementação  
"decorada"

```
@Inject
private ReportService report;

public void showReport(){
    System.out.println(
        report.formatar()
    );
}
```

saída decorada

```
Relatorio Texto
      Usando CDI Decorator
Dados relatorio
```

# 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 {
```

## Managed Beans @Named

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



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

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

# 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 deviam ser feitas manualmente agora temos `@Transactional` na CDI.

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