01

Injetando dependências

Transcrição

Agora que já conseguimos fazer o CDI funcionar dentro da nossa aplicação, vamos utilizá-lo agora para resolver os problemas de acoplamento que temos nas nossas classes.

A classe `LoginBean`, está instanciando `UsuarioDAO`, e já vimos anteriormente o problema de se acoplar com a classe `UsuarioDAO`.

```
boolean existe = new UsuarioDao().existe(this.usuario);
```

Como a classe `UsuarioDAO` é uma dependência nossa, precisamos injetar essa dependência. Isso é possível porque agora é o CDI que instancia a classe `LoginBean` e passa para o JSF. Fizemos isso ao utilizar as annotations `@Named` e `@RequestScoped`.

Para receber o `UsuarioDAO` via injeção de dependências, vamos criar um atributo da classe e utilizamos a annotation `@Inject` em cima do atributo.

```
@Inject
private UsuarioDao usuarioDao;
```

No trecho onde antes era criada uma instância do `UsuarioDAO`, vamos utilizar o atributo:

```
boolean existe = usuarioDao.existe(this.usuario);
```

A classe ficará assim:

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

    private static final long serialVersionUID = 1L;

    private Usuario usuario = new Usuario();

    @Inject
    private UsuarioDao usuarioDao;

    public Usuario getUsuario() {
        return usuario;
    }

    public String efetuaLogin() {
        System.out.println("fazendo login do usuario " + this.usuario.getEmail());

        FacesContext context = FacesContext.getCurrentInstance();
        boolean existe = usuarioDao.existe(this.usuario);
        if(existe ) {
```

```
            context.getExternalContext().getSessionMap().put("usuarioLogado", this.usuario);
            return "livro?faces-redirect=true";
        }

        context.getExternalContext().getFlash().setKeepMessages(true);
        context.addMessage(null, new FacesMessage("Usuário não encontrado"));

        return "login?faces-redirect=true";
    }

    // método deslogar
}
```

Dessa forma, quando o CDI for instanciar a classe `LoginBean`, ele irá verificar que `UsuarioDao` é uma dependência e vai injetá-la dentro de `LoginBean`. Então, este terá um `UsuarioDao` como um objeto já pronto para utilizar sempre que precisar.

Vamos reiniciar o servidor, ao acessarmos a aplicação no navegador é possível perceber que tudo continua funcionando.

Um dos pontos de injeção de dependências que temos é o atributo. Mas vimos que utilizando orientação a objetos, poderíamos pedir as dependências no construtor, e quem fosse utilizar a classe passaria a dependência pra gente. Pensando nisso, o CDI tem um outro ponto de injeção de dependências, que é o construtor.

Nesse caso movemos a anotação para o construtor, em vez de deixar no campo.

```
private UsuarioDao usuarioDao;

@Inject
public LoginBean(UsuarioDao usuarioDao) {
    this.usuarioDao = usuarioDao;
}
```

Existe uma terceira opção que é utilizar um método inicializador:

```
private UsuarioDao usuarioDao;

@Inject
public void inicializador(UsuarioDao usuarioDao) {
    this.usuarioDao = usuarioDao;
}
```

Apesar de existirem esses três pontos de injeção de dependências, normalmente prefere-se realizar a injeção por meio do construtor. Isso ajuda no momento de testar a aplicação. Pois no momento do teste talvez o container do tomcat não esteja rodando, e por consequência não será possível instanciar o `LoginBean` já com as dependências.

Ao testar o `LoginBean` vamos instanciar manualmente a classe, e em seguida vamos precisar passar a dependência para ele. Como recebemos a dependência no construtor, já seria possível passá-la. Podemos instanciar o objeto `UsuarioDao` para em seguida passar no construtor do `LoginBean`.

Agora vamos verificar os outros pontos que temos de acoplamento. Vamos agora alterar a classe `AutorBean`. Dentro dessa classe, instanciámos várias vezes a classe `DAO`. Vamos declarar o `DAO` como dependência da nossa classe, para que ela seja injetada.

```
    private DAO<Autor> autorDao;

    @Inject
    public AutorBean(DAO<Autor> autorDao) {
        this.autorDao = autorDao;

    }
```

Mas ao reiniciar o servidor, recebemos um erro:

```
WELD-001408: Unsatisfied dependencies for type DAO<Autor> with qualifiers @Default
```

Mas por que será que ele não conseguiu injetar a dependência? Vamos comparar a classe `DAO` com a classe `UsuarioDao`.

A classe `UsuarioDAO` não implementa nenhuma interface, não possui nenhuma anotação e tem o método `existe()`, para verificar se o usuário existe ou não.

```
public class UsuarioDao {

    public boolean existe(Usuario usuario) {

        EntityManager em = new JPAUtil().getEntityManager();
        TypedQuery<Usuario> query = em.createQuery(
                  " select u from Usuario u "
                + " where u.email = :pEmail and u.senha = :pSenha", Usuario.class);

        query.setParameter("pEmail", usuario.getEmail());
        query.setParameter("pSenha", usuario.getSenha());
        try {
            query.getSingleResult();
        } catch (NoResultException ex) {
            return false;
        }

        em.close();

        return true;
    }

}
```

Nossa classe `DAO` é uma classe genérica, que também não implementa nenhuma interface e não possui nenhuma annotation.

O que difere uma da outra, é que a classe `DAO` sobrescreveu o construtor, e recebe um parâmetro. Enquanto `UsuarioDao` não, e por isso possui o construtor padrão, sem parâmetros. Por esse motivo o CDI consegue instanciar o `UsuarioDAO` e não consegue instanciar o `DAO`.

```
public class DAO<T> {

    private final Class<T> classe;
```

```
    public DAO(Class<T> classe) {
        this.classe = classe;
    }

    public void adiciona(T t) {
      // código
    }

    public void remove(T t) {
        // código
    }

    public void atualiza(T t) {
        // código
    }

    public List<T> listaTodos() {
        // código
    }

    public T buscaPorId(Integer id) {
        // código
    }

    public int contaTodos() {
        // código
    }

}
```

Precisamos passar essa um objeto do tipo `Class` para que o `DAO` funcione. No método `buscaPorId()`, precisamos do `Class`, pois no ao utilizar o método `find()` passamos esse objeto.

```
public T buscaPorId(Integer id) {
    EntityManager em = new JPAUtil().getEntityManager();
    T instancia = em.find(classe, id); // passando o Class
    em.close();
    return instancia;
}
```

O CDI não vai conseguir criar o `DAO` porque a classe não possui o construtor padrão sem argumentos. Precisamos ensinar o CDI como criar esse objeto. Para fazer isso, vamos criar uma nova classe, no pacote `br.com.alura.livraria.util`, que irá produzir os nossos `DAOs`.

Dentro dessa classe, precisamos ter uma método que irá retornar um DAO genérico. Dentro deste, precisamos instanciar um `DAO` e passar a classe para ele. Ao instanciar o `DAO`, precisamos passar o objeto do tipo `Class` no seu construtor.

```
public class DAOFactory {

    public <T> DAO<T> factory() {
```

```
        return new DAO<T>(classe); // não compila, falta classe
    }
}
```

Perceba que a classe que passamos no construtor de `DAO` é justamente o parâmetro do nosso Generics. Se temos um `DAO<Autor>`, o tipo `Class` que queremos passar é `Autor`. Portanto precisamos pegar esse parâmetro. Para fazer isso, o CDI tem um objeto que representa o ponto onde estamos injetando a dependência (*Injection Point*).

No `AutorBean`, vamos ter um objeto que representa esse ponto de injeção, o construtor. No ponto de injeção temos o objeto que está sendo injetado, em que classe está sendo injetado, entre outras informações. Para fazer uso desse objeto, vamos recebê-lo como parâmetro no método.

Dentro do objeto `InjectionPoint`, é possível obter o tipo que estamos injetando utilindo o método `getType()`. Esse método retorna um `Type` do pacote `import java.lang.reflect`. O `Type` retornado nesse caso será `DAO`, que é o tipo que estamos injetando na minha classe.

```
public <T> DAO<T> factory(InjectionPoint point) {

  Type type = point.getType();

  return new DAO<T>(classe); // não compila, falta classe
}
```

O que precisamos é o argumento que passamos dentro desse tipo, no Generics. Para que isso seja possível, em vez de apenas obter o tipo, vamos fazer um *cast* para `ParameterizedType`. Tendo agora esses tipos que foram parametrizados, poderíamos ter mais de um Generics. No caso, temos apenas um.

O `ParameterizedType` possui um métodos que retorna todos os argumentos, chamado `getActualTypeArguments()`. Nesse caso queremos o primeiro argumento, então, pegar o índice 0. Mais uma vez, vamos receber um `Type`, mas queremos um `Class`, portanto vamos fazer um *cast*.

```
public <T> DAO<T> factory(InjectionPoint point) {

    ParameterizedType type = (ParameterizedType) point.getType();

    Class<T> classe = (Class<T>) type.getActualTypeArguments()[0];

    return new DAO<T>(classe);
}
```

Para que a IDE não mostre o *Warning* pelo fato estarmos fazendo o *cast*. Vamos levar o cursor até a linha do *cast* para *Class*, utilizar o atalho "Ctrl + 1", e escolher a opção "Add @SupressWarnings 'unchecked' to classe". Em seguida vamos mover a linha gerada pelo eclipse para cima da declaração da classe.

```
@SuppressWarnings("unchecked")
public class DAOFactory {
  // restante do código
}
```

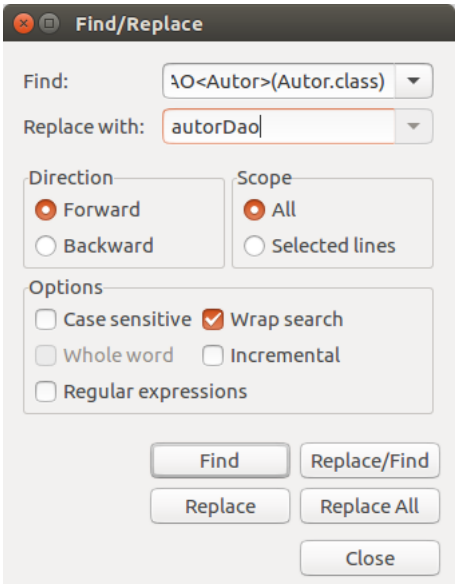
O último ponto é indicar para o CDI que, quando ele precisar de um objeto `DAO`, ele deverá utilizar o método que criamos para produzir. Para isto, anotamos o método com `@Produces`.

```
@SuppressWarnings("unchecked")
public class DAOFactory {

    @Produces // annotation adicionada
    public <T> DAO<T> factory(InjectionPoint point) {

        ParameterizedType type = (ParameterizedType) point.getType();

        Class<T> classe = (Class<T>) type.getActualTypeArguments()[0];

        return new DAO<T>(classe);
    }
}
```

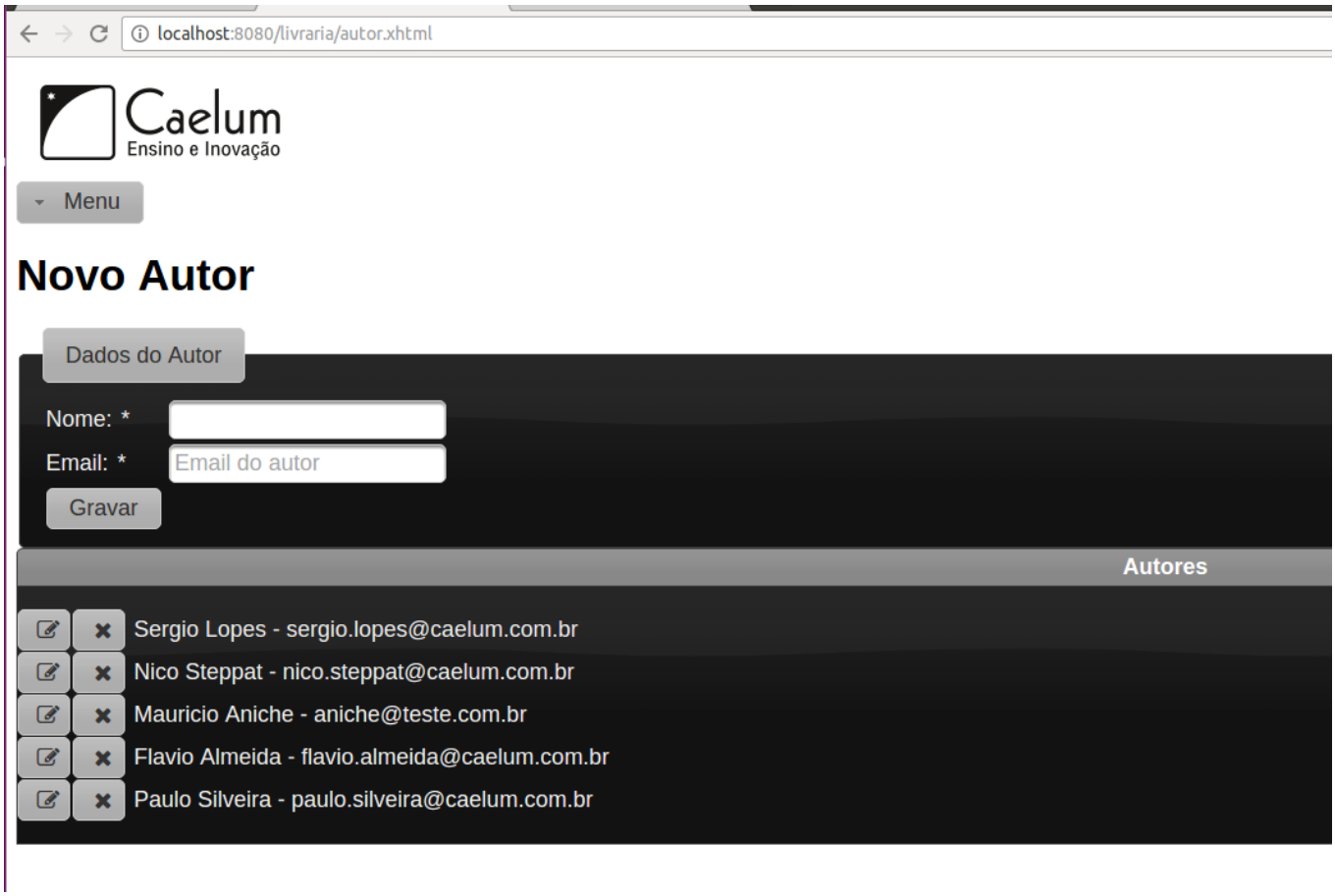
Feito isto, já podemos injetar o nosso `Dao<Autor>`. Agora em vez de ficar instanciarmos o `Dao<Autor>`, vamos utilizar o atributo adicionado que recebemos via injeção de dependências.

Vamos utilizar o Eclipse para substituir as ocorrências. Com a classe `AutorBean` aberta, selecione algum trecho de código que contenha `new DAO<Autor>(Autor.class)`:

Após selecionar, pressione "Ctrl + F" e no campo "Replace with:" digite "autorDao" e em seguida clique em "Replace All". Em seguida clique em "Close".



Agora vamos reiniciar o servidor e verificar se está tudo funcionando. Ao fazer login e acessar a tela de livros, vamos clicar em "ou cadastrar novo autor". Essa tela faz uso do `Dao` e é possível ver que a listagem está funcionando. Se fizermos outras operações, como remover um autor ou cadastrar um novo, é possível ver que tudo funciona.



O próximo passo é receber a dependência nas demais classes. Vamos abrir a classe `LivroBean`, que utiliza o `DAO<Livro>` e o `DAO<Autor>`. Vamos fazer da mesma forma que fizemos no `AutorBean`.

```
private DAO<Autor> autorDao;

private DAO<Livro> livroDao;

@Inject
public LivroBean(DAO<Autor> autorDao, DAO<Livro> livroDao) {
    this.autorDao = autorDao;
    this.livroDao = livroDao;
}
```

Da mesma forma que fizemos anteriormente, vamos fazer agora. Utilizar o "Ctrl + F do Eclipse para substituir as ocorrências de `new DAO<Livro>(Livro.class)` por `livroDao` e `new DAO<Autor>(Autor.class)` por `autorDao`.

Ao reiniciar o servidor e atualizar a página no navegador, recebemos um erro indicando que não é possível produzir um objeto que não seja `Serializable`:

```
WELD-000054: Producers cannot produce unserializable instances for injection into an injection point
```

Para resolver vamos fazer a classe `DAO` implementar a interface `Serializable`. Mova o cursor o nome da classe (que estará sublinhada de amarela) e pressione "Ctrl + 1" ou clique na lâmpada do *Warning* do Eclipse. Escolha a opção "Add generated serial version ID".

```
public class DAO<T> implements Serializable {

    private static final long serialVersionUID = -1603139969274284275L;

    // restante do código
}
```

Ao reiniciar a aplicação, tudo funciona. Agora injetamos nossos DAOs em todos os `bean`s. Em seguida, vamos ver como podemos fazer para resolver outras dependências, como as dependências do `DAO`.