07

Mãos a obra: Débitos Técnicos

Quando efetuamos o desenvolvimento da nossa biblioteca `Alura-lib`, deixamos alguns débitos técnicos passarem.

E chegou a hora de pagarmos esses débitos.

Olhando nossa classe `DAO` temos dois débitos técnicos

- Estamos pressupondo que quem utilizar nosso `DAO`, vai modelar suas classes utilizando o `ID` com o tipo `Integer`.
- Estamos fazendo a contagem somente dos livros, e queremos contar os dados da entidade que estamos trabalhando.

Vamos resolver esse débitos agora para isso vamos fazer as seguintes alterações:

- Receber um novo argumento genérico para o tipo do id
- Converter a query para a contagem com `Criteria`

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

    ...

    public T buscaPorId(I id) {
        T instancia = em.find(classe, id);
        return instancia;
    }

    public Long contaTodos() {

        CriteriaBuilder builder = em.getCriteriaBuilder();
        CriteriaQuery<Long> query = builder.createQuery(Long.class);

        query.select(builder.count(query.from(classe)));


        Long result = em.createQuery(query).getSingleResult();

        return result;
    }
}
```

Quando fizemos essa alteração quebramos nossa fabrica de `DAO`, vamos corrigi-lo informando o segundo argumento genérico:

```
public class DAOFactory {

    @Inject
    private EntityManager manger;

    @Produces
    public <T,I> DAO<T,I> factory(InjectionPoint point){
```

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

        Class<T> classe = (Class<T>) type.getActualTypeArguments()[0];
        return new DAO<T,I>(classe, manger);
    }

}
```

Vamos instalar nosso `jar` no repositório local: Clique com o botão direito do mouse sobre o projeto `Alura-lib` e selecione: `Run as > Maven Install` .

No projeto `Livraria` vamos atualizar as dependências do `maven` : Clique com o botão direito do mouse sobre o projeto `Livraria` e selecione `Maven > Update project...` .

Quando atualizamos nosso projeto vimos que nossos `Bean` s quebraram, pois não informamos o segundo argumento genérico para injetar o `DAO` . Para corrigir isso basta informar o segundo argumento genérico nos pontos de injeções do `DAO` (exemplo: `DAO<Livro, Integer>`).

Agora vamos analisar a classe `JPAFactory` , nela estamos pressupondo que quem utilizar nossa biblioteca vai ter configurado uma `persistence-unit` com o nome `livraria` no arquivo `persistence.xml` .

Vamos implementar um arquivo de configuração para nossa biblioteca. A ideia é que, quem utilizar nossa biblioteca `Alura-lib` tenha no seu `classpath` um arquivo chamado `alura-lib.properties` e a partir desse arquivo vamos criar um objeto `Properties` e utilizar onde for necessário.

Vamos criar a classe que efetua a leitura do arquivo `alura-lib.properties` :

```
public class ConfigurationFactory {

    @Produces
    @Configuration
    @ApplicationScoped
    public Properties getProperties() throws IOException{
        InputStream inputStream = ConfigurationFactory.class.getResourceAsStream("/alura-lib.properi

        Properties properties = new Properties();
        properties.load(inputStream);

        return properties;
    }


}
```

Perceba que estamos produzindo nosso objeto `Properties` com o escopo de aplicação, para que só seja necessário efetuar a leitura dele uma única vez. Também criamos um qualificador para que não gere conflito caso alguém precise produzir um objeto `Properties` .

```
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Configuration {}
```

Vamos agora alterar nossa classe `JPAFactory` para que ela utilize nossa `Properties` e crie um `EntityManagerFactory` que foi configurado no arquivo `alura-lib.properties` .

```
ApplicationScoped
public class JPAFactory {

    private EntityManagerFactory emf;

    @Inject    @Configuration
    private Properties properties;


    @Produces
    @RequestScoped
    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public void close(@Disposes EntityManager em) {
        if (em.isOpen()) {
            em.close();
        }
    }

    @PreDestroy
    public void preDestroy(){
        if (emf.isOpen()) {
            emf.close();
        }
    }


    @PostConstruct
    public void loadEMF(){

        emf = Persistence
            .createEntityManagerFactory(properties.getProperty("alura.lib.persistenceUnit"));


    }

}
```

Pronto agora quem utilizar nossa biblioteca, basta definir um arquivo `alura-lib.properties` e nele definir uma chave `alura.lib.persistenceUnit` com o nome da `persistence-unit` que deve ser utilizado.

Nosso último débito técnico está na classe `PhaseListernerObserver` , essa classe só deve ser utilizada pela classe `PhaseListernerGenerico` . Porém da forma que está atualmente qualquer um pode injetar um objeto de `PhaseListernerObserver` para impedir que isso ocorra, vamos informar ao **CDI** que essa classe nunca deve ser injetada.

Para isso vamos anotar a classe `PhaseListernerObserver` com `@Vetoed` .

Feito tudo isso vamos instalar nosso `jar` no repositório local: Clique sobre o projeto `Alura-lib` e selecione `Run as > Maven Install` .

Depois vamos atualizar o projeto `Livraria` : Clique com o botão direito do mouse sobre o projeto `Livraria` e selecione `Maven > Update project...` .

Crie o arquivo necessário para que nossa biblioteca funcione. No projeto `Livraria` crie o arquivo `src/main/resources/alura-lib.properties`

```
alura.lib.persistenceUnit=livraria
```

Rode o projeto `Livraria` novamente e verifique se tudo continua funcionando.