02

# Débito técnico

### Transcrição

Vamos verificar quais são os débitos técnicos que temos no projeto `alura-lib`.

Começando pela classe `DAO`, no pacote `br.com.alura.alura_lib.dao`. Temos uma classe genérica, mas no método `buscaPorId()`, recebemos um `Integer` como argumento. Quem utilizar a classe pode querer passar um `Long`, por exemplo.

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

Como não sabemos qual tipo será passado, vamos receber um tipo genérico:

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

É necessário indicar o tipo `I` na declaração da classe:

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

Dessa forma o `T` representa o tipo da classe e o `I` o tipo do `id`. Como foi feita essa alteração no pacote `br.com.alura.alura_lib.factory`, a classe `DAOFactory` parou de funcionar, porque o método `factory()` espera que o `DAO` trabalhe apenas com um tipo genérico:

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

Precisamos passar os dois tipos:

```
@Produces
public <T, I> DAO<T, I> factory(InjectionPoint point) { // passando o segundo tipo genérico

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

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

    return new DAO<T, I>(classe, manager); // passando o segundo tipo genérico
}
```

Vamos instalar a biblioteca no repositório local e, depois, vamos atualizar o projeto `livraria`.

Com a atualização, algumas classes ficaram quebradas - justamente as que utilizam o `DAO`:


Isto ocorreu porque estamos passando apenas uma referência para um tipo genérico, mas agora é necessário passar duas. Vamos iniciar as alterações a partir da classe `AutorBean`:

```
@Model
public class AutorBean implements Serializable {

    // outros atributos

    private DAO<Autor, Integer> autorDao;

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

    // restante do código
}
```

Foi adicionado o tipo `Integer`, pois esse é o tipo utilizado para armazenar o `id` do `Autor`. Vamos fazer as mesmas alterações para as outras classes. Agora a classe `LivroBean`:

```
@ViewModel
public class LivroBean implements Serializable {

    // outros atributos

    private DAO<Autor, Integer> autorDao;

    private DAO<Livro, Integer> livroDao;

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

    // restante do código
}
```

E por fim, a classe `VendasBean`:

```
@ViewModel
public class VendasBean implements Serializable{

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

```
    private DAO<Livro, Integer> livroDao;

    public VendasBean(DAO<Livro, Integer> livroDao) {
        this.livroDao = livroDao;
    }

    // restante do código
}
```

Ao reiniciar o servidor e acessar o sistema, passando pelas suas funcionalidades - como inserir um livro, editar e remover -, e fazer essas mesmas operações com os autores, tudo continuou funcionando.

Ainda na classe `DAO`, temos o método `contarTodos()`:

```
public int contaTodos() {
    long result = (Long) em.createQuery("select count(n) from livro n")
            .getSingleResult();

    return (int) result;
}
```

O método faz uma *query* específica em `livro`. Gostaríamos que o `contarTodos()` conte os registros da entidade que estamos utilizando e não funcione apenas com livros.

Para fazer essa *query*, vamos utilizar `Criteria`. Foi criado o `CriteriaBuilder` e, em seguida, o `CriteriaQuery`. O `Long` na `query` indica a classe que será retornada ao realizar a *query*.

```
public int contaTodos() {

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

        // restante do código
}
```

Agora podemos utilizar o método `select` e como argumento fazer uma chamada ao método `count()`, do `builder`. Passamos para o `count` a classe na qual queremos fazer a *query*.

Em seguida, basta chamar o método `createQuery()` do `em`, utilizar o `getSingleResult()` e retornar o `Long`. O retorno do método foi alterado de `int` para `Long`:

```
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;
    }
```

Vamos instalar a biblioteca no repositório local e atualizar o projeto `livraria`. Isso não gerou nenhum erro. Em seguida damos um "Clean" e reiniciamos o Tomcat. Ao acessar as funcionalidades do sistema, tudo continua funcionando.

Não temos mais o que fazer na classe `DAO`. Vamos agora analisar a classe `DAOFactory`, do pacote `br.com.alura.alura_lib.factory`. A classe apenas instancia um novo `DAO` passando os parâmetros obtidos no ponto de injeção. Tudo certo com essa classe:

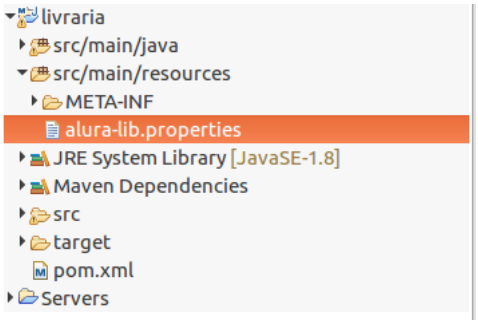
```
@SuppressWarnings("unchecked")
public class DAOFactory {

    @Inject
    private EntityManager manager;

    @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, manager);
    }
}
```

## Obtendo a persistence-unit a partir de um arquivo

Na classe `JPAFactory`, ainda no pacote `br.com.alura.alura_lib.factory`, temos um outro débito técnico. Estamos criando o `EntityManagerFactory` assumindo que o nome da `persistence-unit` no arquivo `persistence.xml` será sempre `livraria`:

```
private EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("livraria");
```

Para resolver esse problema, podemos assumir que quem for utilizar a biblioteca, terá dentro do diretório `src/main/resources` o arquivo `alura-lib.properties`:




Dentro do arquivo, seria definido o nome da `persistence-unit`:

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

Na classe `JPAFactory`, podemos utilizar o arquivo. Em vez de instanciarmos o `EntityManagerFactory` diretamente no atributo, vamos extrair isso para um método e utilizar mais um *callback* do *interceptor*, para que a atribuição seja feita após a criação da classe.

É preciso obter o arquivo a partir do *classpath*, para isso utilizamos o seguinte código:

```
InputStream inputStream = JPAFactory.class.getResourceAsStream(";alura-lib.properties");
```

Em seguida utilizamos a classe `Properties` do java, onde temos uma sobrecarga do método `load()` que recebe um `InputStream` e carrega o arquivo:

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

Agora é possível ler uma propriedade a partir de uma chave, que é justo a chave que definimos no arquivo `alura.lib.persistenceUnit`:

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

Desta forma é possível utilizar a *properties*. Mas pensando na evolução do nosso software, muito provavelmente serão adicionadas novas configurações no arquivo `alura-lib.properties`, porque por meio de *properties* é possível um melhor reuso da biblioteca - quando escrever uma propriedade no arquivo e alterar o comportamento dessa biblioteca.

Para evitar a todo momento termos que instanciar um objeto do tipo `Properties`, ou obter um recurso do *classpath*, vamos extrair esse comportamento para uma nova classe. Será a classe `ConfigurationFactory`, que ficará no pacote `br.com.alura.alura_lib.configuration`:

```
package br.com.alura.alura_lib.configuration;

public class ConfigurationFactory {

}
```

A classe irá produzir uma `Properties`:

```
public class ConfigurationFactory {

    @Produces
    public Properties getProperties() throws IOException {
        InputStream inputStream = ConfigurationFactory.class.getResourceAsStream(";alura-lib.properti

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

        return properties;
    }
}
```

O método `load()` lança uma exceção, mas como não estamos interessados em tratá-la, vamos simplesmente utilizar o `throws`.

Pode acontecer de outras pessoas produzirem um objeto do tipo `Properties`. Para que não haja conflitos, vamos criar um qualificador. A anotação `Configuration` será criada no pacote `br.com.alura.alura_lib.configuration.annotation`.

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

}
```

Na classe `ConfigurationFactory`, basta utilizar o qualificador. Já que vamos ler as propriedades uma única vez e utilizar em toda a aplicação, vamos também definir o escopo como sendo `@ApplicationScoped`:

```
public class ConfigurationFactory {

    @Produces
    @Configuration
    @ApplicationScoped
    public Properties getProperties() throws IOException {
      // código do método
    }
}
```

Agora é necessário receber uma `Properties` via injeção de dependências, na classe `JPAFactory`. É necessário atribuir o `EntityManagerFactory` para a propriedade `emf`, dentro do método `loadEmf()`.

```
@ApplicationScoped
public class JPAFactory {

    private EntityManagerFactory emf;

    @Inject @Configuration
    private Properties properties;

    // restante do código

    @PostConstruct
    public void loadEMF() {
        emf = Persistence.createEntityManagerFactory(properties.getProperty("alura.lib.persistenceUr
    }
}
```

Vamos realizar os passos de sempre: instalar a biblioteca no repositório local, atualizar o projeto `livraria`, dar um "Clean" e reiniciar o Tomcat. Ao testar o sistema, tudo funciona.

Em algumas classes da biblioteca não temos mais trabalho a fazer. Para que ninguém seja capaz de instanciar a classe `PhaseListenerObserver`, do pacote `br.com.alura.alura_lib.jsf.phaselistener`, vamos marcá-la com a anotação `@Vetoed`:

```
@SuppressWarnings("serial")
@Vetoed
public class PhaseListenerObserver {
```

Poderíamos ir um pouco mais além e em vez de usar o `@Vetoed`, retirar o modificador `public` da classe. Desta forma, apenas quem estiver dentro do mesmo pacote que essa classe conseguiria instanciá-la:

```
@SuppressWarnings("serial")
class PhaseListenerObserver {
```

Neste caso, vamos apenas anotar a classe com `@Vetoed`.

Nós conseguimos deixar a biblioteca sem débitos técnicos e, então, podemos utilizá-la nos nossos projetos.

Espero que você tenha gostado do curso, e se tiver dúvidas, não deixe de usar o fórum.