02

Incrementando lib

Transcrição

Vamos ver agora o que conseguimos extrair de funcionalidades do projeto `livraria` para nossa *lib*.

Talvez quem for utilizar o `GerenciadorDeTransacao` não queira que o *interceptor* esteja habilitado por padrão. Vamos permitir que quem for utilizar o `GerenciadorDeTransacao` decida se ele deve estar habilitado ou não. Para isso vamos retirar a anotação `@Priority`.

```
@Interceptor
@Transacional
public class GerenciadorDeTransacao implements Serializable {
```

Dessa forma quem utilizar a biblioteca pode habilitar no arquivo `beans.xml`. No projeto `livraria`, vamos remover os comentários do arquivo `beans.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/bea
       version="1.2" bean-discovery-mode="all">

    <interceptors>
        <class>br.com.alura.alura_lib.tx.GerenciadorDeTransacao</class>
    </interceptors>
</beans>
```

Dentro da classe `JPAFactory`, não estamos fechando o `EntityManagerFactory`. Já que temos o método `close()` seria interessante utilizá-lo. Para isso temos o *event callback* chamado `@PreDestroy`. Antes do CDI descartar a instância, passamos pelo *callback*.

Fazemos uma verificação similar ao que fazemos no método `close()` que fecha o `EntityManager`:

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

O problema é que como não definimos escopo para a `JPAFactory`, sempre que o CDI precisar produzir um objeto ele dará um `new` na classe. A cada requisição, quando o `EntityManager` for destruído, o `JPAFactory` também será destruído. Dessa forma ele irá sempre invocar o método `PreDestroy`, o que fará com que tenhamos erros na aplicação.

O ideal é que a `JPAFactory` fique no escopo de aplicação. Desta forma será criado um objeto da classe que irá durar a aplicação inteira. Para isso anotamos a classe como `@ApplicationScoped` (pacote

`javax.enterprise.context.ApplicationScoped`).

Como é uma classe de aplicação, não necessitamos mais do atributo estático, pois só teremos uma instância da classe por aplicação:

```
@ApplicationScoped
public class JPAFactory {

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

    // restante do código
}
```

Refatorando o LoginBean

No projeto `livraria` , vamos analisar o método `efetuaLogin()` , na classe `LoginBean` :

```
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";
}
```

O método está com algumas responsabilidades a mais. Como obter o `FacesContext` :

```
FacesContext context = FacesContext.getCurrentInstance();
```

Dado o `FacesContext` , tem que pegar o contexto externo, em seguida obter o `Map` da sessão para adicionar o usuário:

```
context.getExternalContext().getSessionMap().put("usuarioLogado", this.usuario);
```

Além de saber detalhes do que precisa para manter as mensagens no escopo de *flash*, é adicionada uma mensagem no contexto:

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

Então temos algumas coisas que fogem da responsabilidade do método `efetuaLogin()`. Portanto vamos extrair algumas coisas.

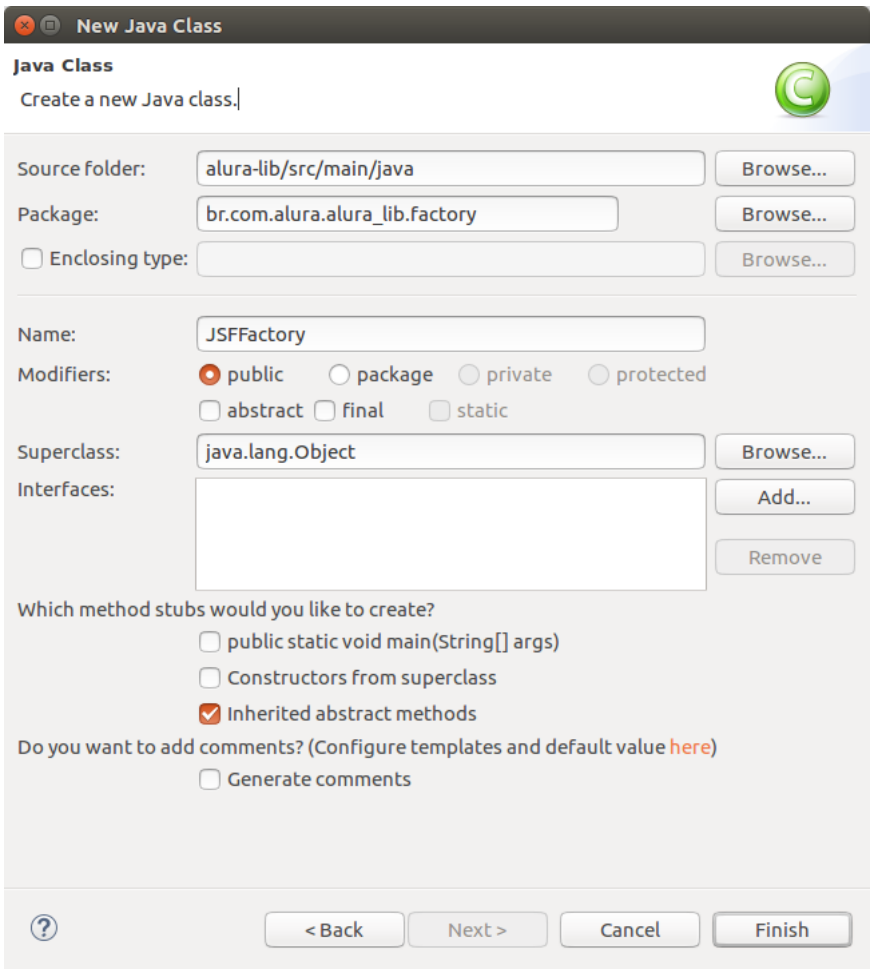
O `FacesContext` é uma dependência nossa, que utilizamos em vários momentos, listados anteriormente. Então vamos receber esse objeto no construtor:

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

       // outros atributos

    private FacesContext context;

    @Inject
    public LoginBean(UsuarioDao usuarioDao, FacesContext context) {
        this.usuarioDao = usuarioDao;
        this.context = context;
    }

    // restante do código
}
```

Dessa forma, podemos **remover** a linha que cria o `FacesContext` no método `efetuaLogin():

```
FacesContext context = FacesContext.getCurrentInstance();
```

O `FacesContext` é uma classe abstrata, portanto o CDI não irá conseguir instanciar um objeto dessa classe. Vamos ter que ensinar ao CDI como ele pode produzir o `FacesContext`. No `alura-lib`, vamos criar uma classe chamada `JSFFactory`, no pacote `br.com.alura.alura_lib.factory`.



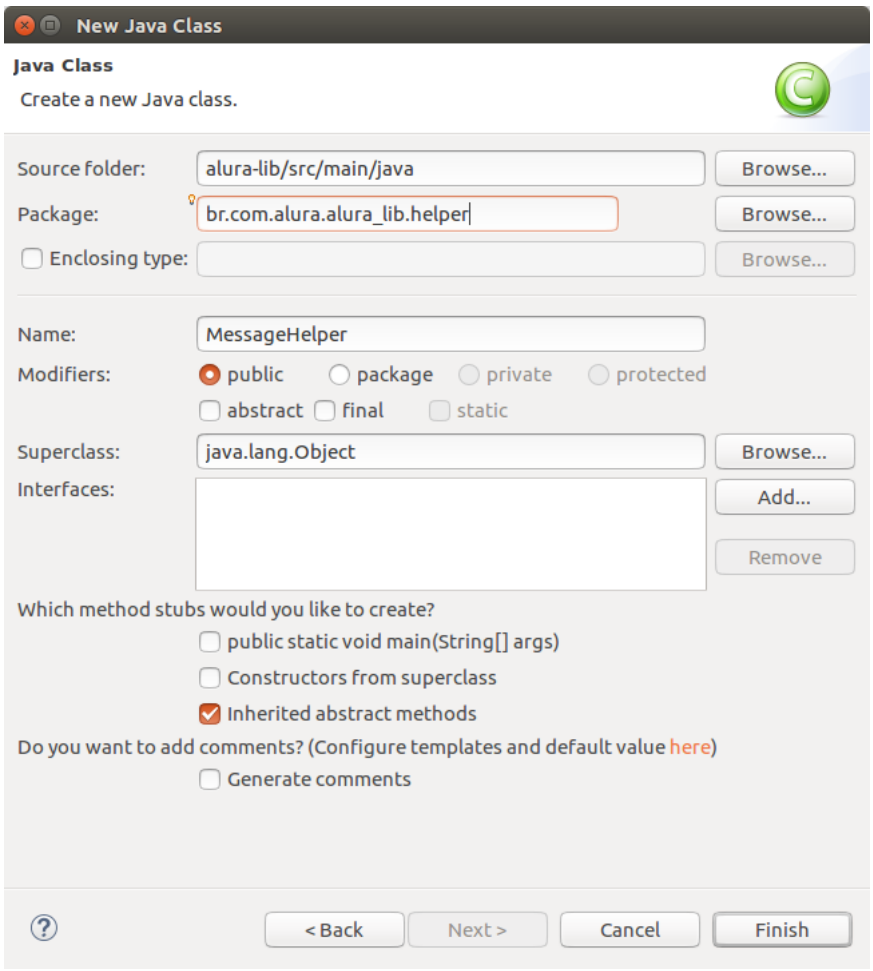
A classe terá um método que retorna um `FacesContext` e vai utilizar o método estático `getCurrentInstance()` para obter a instância do `FacesContext`. Vamos marcar com a anotação `@Produces`, para indicar que aquele é o produtor de `FacesContext`. Como as informações do `FacesContext` mudam a cada *request*, vamos definir o escopo como `RequestScoped`:

```
public class JSFFactory {

    @Produces
    @RequestScoped
    public FacesContext getFacesContext() {
        return FacesContext.getCurrentInstance();
    }
}
```

Vamos retornar ao `LoginBean` e tentar resolver os outros problemas. Adicionar uma mensagem é algo comum que vamos utilizar em outros pontos da aplicação. Além disso, será complexo se quisermos que ela fique no escopo de *flash*, temos um parâmetro que estamos passando `null`. Vamos buscar uma forma de encapsular a complexidade.

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

Dentro da biblioteca, vamos criar um *helper*. Uma classe chamada `MessageHelper` criada dentro do pacote `br.com.alura.alura_lib.helper`.



A classe possui o método `addMessage()` que recebe o `FacesMessage` e uma sobrecarga, que recebe o `clientId` e a mensagem. Recebemos o `FacesContext` via injeção de dependências:

```
public class MessageHelper {

    private FacesContext context;

    @Inject
    public MessageHelper(FacesContext context) {
        this.context = context;

    }

    public void addMessage(FacesMessage message) {
        addMessage(null, message);
    }

    private void addMessage(String clientId, FacesMessage message) {
        context.addMessage(clientId, message);
    }
}
```

Falta agora resolver a questão de adicionar a mensagem no escopo de flash. Vai ter momento que vamos precisar disso e outros momentos que não. Vamos adicionar um método no `MessageHelper`:

```
public MessageHelper onFlash() {
    context.getExternalContext().getFlash().setKeepMessages(true);
```

```
        return this;
    }
```

O método possui uma interface fluente, dessa forma podemos habilitar o manter a mensagem de *flash* e em seguida ao retornar o próprio objeto, podemos realizar a chamada ao método `addMessage()`.

Vamos instalar a biblioteca no repositório local e em seguida rodar o "Maven > Update Project" na `livraria`.

No `LoginBean` vamos recebr o `MessageHelper`:

```
private MessageHelper helper;

@Inject
public LoginBean(UsuarioDao usuarioDao, FacesContext context, MessageHelper helper) {
    this.usuarioDao = usuarioDao;
    this.context = context;
    this.helper = helper;
}
```

Vamos **remover** as linhas do método `efetuaLogin()`:

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

E utilizar o *helper*:

```
public String efetuaLogin() {

    // restante do código

    helper
            .onFlash()
            .addMessage(new FacesMessage("Usuário não encontrado"));

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

Vamos dar um *clean* e iniciar/reiniciar o Tomcat. Ao tentar logar com um usuário inválido, recebemos a mensagem de erro, como esperado. E ao inserir o usuário certo, fomos direcionados para a página de `livro`. Tudo está funcionando corretamente.