03

Qualificadores

Transcrição

Ainda tem algo que incomoda na classe `MessageHelper`:

No método `onFlash()`, temos o contexto (`context`), em seguida, pegamos o contexto externo,e depois pegar o `flash`. Só então definiremos que a mensagem deve durar o escopo de *flash*.

```
context.getExternalContext().getFlash().setKeepMessages(true);
```

Não precisamos saber de detalhes, como o que pegamos o *flash* a partir do contexto externo. Simplesmente precisamos do *flash*. É uma dependência da classe `MessageHelper`. Já que é uma dependência, vamos receber via injeção de dependências, e criar um atributo chamado `flash`.

```
public MessageHelper onFlash() {
    flash.setKeepMessages(true);
    return this;
}
```

Vamos receber o `flash` via injeção de dependências no construtor:

```
public class MessageHelper {

    private FacesContext context;
    private Flash flash;

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

    }

    // restante do código
}
```

O `Flash`, do pacote `javax.faces.context.Flash`, é uma classe abstrata e portanto o CDI não conseguirá injetar. Precisamos criar um produtor, que será adicionado na classe `JSFFactory`:

```
@Produces
@RequestScoped
public Flash getFlash() {
    return getFacesContext().getExternalContext().getFlash();
}
```

Vamos instalar a biblioteca no repositório local, utilizar o atalho "Ctrl + F5" no projeto `livraria` para que o Maven atualizar o `jar` da lib, dar um *clean* e reiniciar o Tomcat.

O projeto continua funcionando e agora nós temos uma classe mais coesa. Agora vamos voltar a resolver problemas no nosso `LoginBean`.

No método `efetualLogin()`, temos que pegar o contexto, obter o contexto externo, obter o mapa da sessão para em seguida adicionar o usuário nesse mapa.

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

Algo similar ocorre no método `deslogar()`, porém removendo o usuário do mapa:

```
context.getExternalContext().getSessionMap().remove("usuarioLogado");
```

A ideia é removermos esse código. Já que precisamos do mapa de sessão, vamos recebê-lo como uma dependência. O método `getSessionMap()` é um método abstrato na classe `ExternalContext`:

```
public abstract Map<String, Object> getSessionMap();
```

Vamos adicionar um método produto que retorna um `Map` desse tipo. O método será adicionado na classe `JSFFactory`.

O código para obter o *session map* é o seguinte:

```
return getFacesContext().getExternalContext().getSessionMap();
```

e o código para obter o *flash* é o seguinte:

```
return getFacesContext().getExternalContext().getFlash();
```

Vamos extrair o código repetido para um método privado:

```
private ExternalContext.getExternalContext() {
    return getFacesContext().getExternalContext();
}
```

Agora podemos reutilizar o método nos produtores:

```
@Produces
public Map<String, Object> sessionMap() {
    return getExternalContext().getSessionMap();
}
```

Vamos aproveitar e reutilizar no `getFlash()`:

```
@Produces
@RequestScoped
public Flash getFlash() {
    return getExternalContext().getFlash();
}
```

Agora que queremos receber o `SessionMap` por injeção de dependências, vamos querer fazer algo assim, nos métodos `efetuaLogin()` e `deslogar()`:

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

    boolean existe = usuarioDao.existe(this.usuario);
    if(existe) {

        sessionMap.put("usuarioLogado", this.usuario); // atributo injetado

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

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

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

public String deslogar() {

    sessionMap.remove("usuarioLogado"); // atributo injetado

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

Agora que fizemos as devidas alterações, não precisamos mais receber o `FacesContext`, portanto vamos remover o `FacesContext` e adicionar o mapa da sessão.

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

    private static final long serialVersionUID = 1L;

    private Usuario usuario = new Usuario();

    private UsuarioDao usuarioDao;

    private MessageHelper helper;

    private Map<String, Object> sessionMap;

    @Inject
    public LoginBean(UsuarioDao usuarioDao, MessageHelper helper, Map<String, Object> sessionMap) {
        this.usuarioDao = usuarioDao;
```

```
        this.helper = helper;
        this.sessionMap = sessionMap;
    }

    // restante do código
}
```

Vamos seguir aqueles velhos passos que já estamos acostumados: instalar a biblioteca no repositório local, *update project* no projeto `livraria`, e dar um *clean* e reiniciar o Tomcat, para verificar se tudo funcionou.

Mas recebemos um erro:

```
WELD-001409: Ambiguous dependencies for type Map<String, Object> with qualifiers @Default
```

Na mensagem de erro, o Weld informa que temos dependência ambíguas, com o qualificador `@Default`. Ele também informa quais são as dependências que estão gerando o mesmo tipo:

```
Possible dependencies:
  - Producer Method [Map<String, Object>] with qualifiers [@Any @Default] declared as [[BackedAnnota
  - Producer Method [Flash] with qualifiers [@Any @Default] declared as [[BackedAnnotatedMethod] @Pr
```

O primeiro é esperado. Foi o produtor que criamos para que fosse possível injetar o `Map<String, Object>`. Mas e o método `getFlash()`? O `Flash`, tipo retornado por esse método, implementa a interface `Map`:

```
public abstract class Flash implements Map<String, Object> {
```

Dessa forma, o CDI irá atender sempre que alguém precisar de um `Flash` ou de um `Map<String, Object>`, pois podemos dizer que o `Flash` **é** um `Map<String, Object>`, já que ele implementa a interface `Map`.

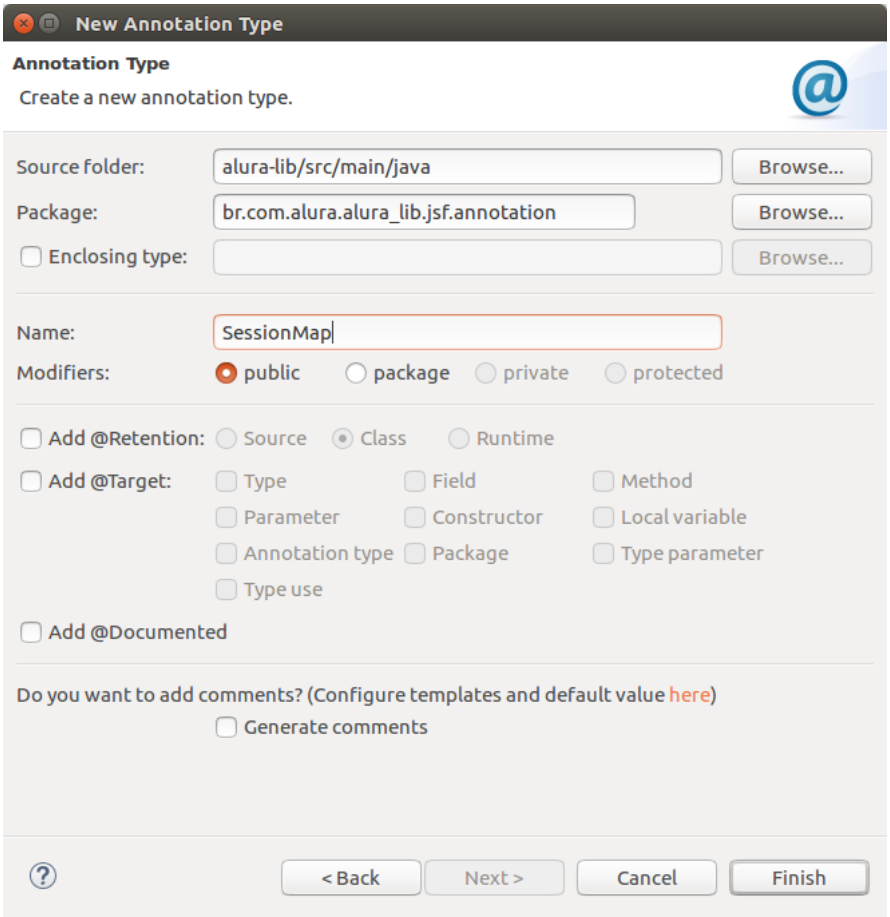
Então no `JSFFactory`, temos duas formas de produzir um `Map<String, Object>`. Nossa intenção é distingui-los. Para resolver isso, o CDI possui qualificadores (*qualifiers*).

Perceba que na mensagem de erro, somos informados que existem dependências ambíguas com o qualificador `@Default`:

```
WELD-001409: Ambiguous dependencies for type Map<String, Object> with qualifiers @Default
```

Quando não definimos nenhum qualificador, o CDI automaticamente colocará o `@Default`. Quando produzimos os objetos, eles já estão sendo produzidos com o qualificador `@Default`, e por isso o conflito entre as dependências.

No projeto `alura-lib`, vamos criar uma *annotation* dentro de um pacote específico para o JSF. O nome será `SessionMap` e o pacote `br.com.alura.alura_lib.jsf.annotation`:



A novidade no código da *annotation* em relação a que foi criada quando estudamos *interceptors*, é que devemos utilizar a anotação `@Qualifier`, para indicar que a anotação é um *qualifier*. O `ElementType.PARAMETER` indica que podemos utilizar a anotação junto com parâmetros, em métodos e construtores. Utilizamos o `ElementType.FIELD` para que seja possível anotar atributos.

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

}
```

No método produtor, em `JSFFactory`, basta utilizar a anotação:

```
@Produces
@SessionMap
public Map<String, Object> sessionMap() {
    return geExternalContext().getSessionMap();
}
```

Vamos instalar a *lib* no repositório local. E rodar um *update project* no projeto `livraria`.

Quando formos injetar a dependência, precisamos indicar que queremos receber a dependência que possui esse qualificador. No construtor da classe `LoginBean`, junto ao `Map<String, Object>`, utilizamos a anotação `@SessionMap`.

```
@Inject
public LoginBean(UsuarioDao usuarioDao, MessageHelper helper, @SessionMap Map<String, Object> sessi
```

```
    this.usuarioDao = usuarioDao;
    this.helper = helper;
    this.sessionMap = sessionMap;
}
```

Vamos dar um *clean* e reiniciar o Tomcat, para ver se tudo funciona. Dessa vez não recebemos erro, pois resolvemos o conflito. Então a aplicação funciona normalmente.

Além de termos o mapa de sessão, temos também no JSF, o mapa de aplicação e o mapa de *request*, que pode ser útil em alguns casos. Vamos deixar os produtores prontos para o caso de alguém querer utilizar:

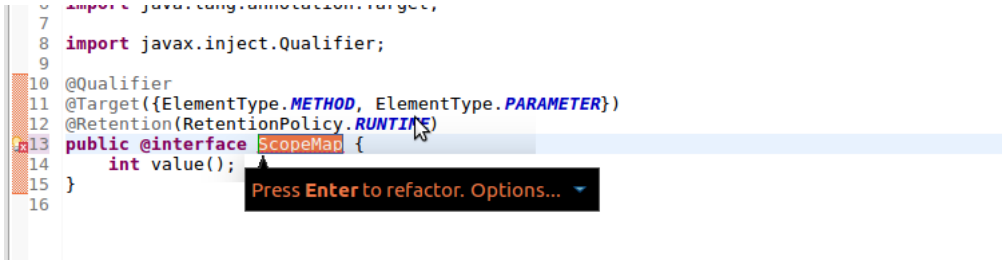
```
@Produces
public Map<String, Object> requestMap() {
    return geExternalContext().getRequestMap();
}

@Produces
public Map<String, Object> applicationMap() {
    return geExternalContext().getApplicationMap();
}
```

Como estamos retornando em todos os casos um `Map<String, Object>`, precisamos utilizar qualificadores. Da forma que fizemos até agora, precisaríamos criar mais duas anotações: uma para o `requestMap()` e outra para o `applicationMap()`.

Mas isso não é necessário. Qualquer atributo que temos dentro da anotação criada também será um qualificador. Antes de mostrar um exemplo, vamos renomear a anotação `SessionMap` para explicitar o nome.

No editor, basta selecionar o nome da anotação, pressionar "Shift + Alt + R" e mudar o nome para `ScopeMap`:



Se criarmos, por exemplo, um `int value()`:

```
@Qualifier
@Target({ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ScopeMap {
    int value();
}
```

No produtor, podemos passar o valor 1, por exemplo. No momento em que utilizarmos o qualificador com o valor 1 no ponto de injeção de dependências, será injetado o mapa da sessão:

```
@Produces
@ScopeMap(1)
public Map<String, Object> sessionMap() {
    return getExternalContext().getSessionMap();
}
```

No `LoginBean`, ficaria da seguinte forma (ainda não funciona porque não instalamos o `jar` localmente):

```
public LoginBean(UsuarioDao usuarioDao, MessageHelper helper, @ScopeMap(1) Map<String, Object> sessi
```

Seguindo com o exemplo, poderíamos definir que:

```
@Produces
@ScopeMap(0)
public Map<String, Object> requestMap() {
    return geExternalContext().getRequestMap();
}

@Produces
@ScopeMap(2)
public Map<String, Object> applicationMap() {
    return geExternalContext().getApplicationMap();
}
```

Dessa forma se utilizarmos o valor 0 recebemos o mapa de *request*, e se utilizarmos o valor 2 recebemos o mapa de aplicação. Então o valor que temos no atributo é utilizado para diferenciar um *bean* do outro.

O que podemos fazer, em vez de utilizar um `int`, é utilizar uma `Enum`. Para este não ficar jogado pela biblioteca, e como ele só fará sentido para a anotação, vamos criar o Enum dentro da anotação:

```
@Qualifier
@Target({ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface ScopeMap {
    ScopeMap.Scope value();

    enum Scope {
        REQUEST, SESSION, APPLICATION
    }
}
```

Agora o `value()` é do tipo `Scope` (a Enum). Dessa forma podemos utilizar os valores nos métodos produtores:

```
@Produces
@ScopeMap(Scope.REQUEST)
public Map<String, Object> requestMap() {
    return geExternalContext().getRequestMap();
}

@Produces
```

```
@ScopeMap(Scope.SESSION)
public Map<String, Object> sessionMap() {
    return geExternalContext().getSessionMap();
}

@Produces
@ScopeMap(Scope.APPLICATION)
public Map<String, Object> applicationMap() {
    return geExternalContext().getApplicationMap();
}
```

Vamos instalar a biblioteca no repositório local, e atualizar o projeto `livraria` (botão direito na parte de cima do projeto, `Maven > Update Project`), para que seja possível utilizar as modificações que fizemos na biblioteca.

Em `LoginBean` vamos utilizar o *qualifier* para receber o mapa de sessão:

```
@Inject
public LoginBean(UsuarioDao usuarioDao, MessageHelper helper, @ScopeMap(Scope.SESSION) Map<String, (
   // código do construtor
}
```

Por fim, vamos dar um *clean* e reiniciar o Tomcat, para verificar se tudo continua funcionando. E sim, tudo continua funcionando perfeitamente!

Com qualificadores, o CDI consegue distinguir *beans* do mesmo tipo. Se temos que produzir um objeto, e temos mais de um objeto produzindo *beans* do mesmo tipo, precisamos de um qualificador para distinguir um objeto do outro.