03

BeanManager

Transcrição

Vimos que como `PhaseListenerGenerico` é um `PhaseListener`, não será possível injetar as dependências, por esta ser gerenciada pelo JSF em vez de ser pelo CDI. Considerando que não receberemos o `PhaseListenerObserver` injetado, vamos instanciá-lo manualmente:

```
private PhaseListenerObserver observer = new PhaseListenerObserver();
```

Porém estamos recendo alguns objetos via injeção de dependências também no `PhaseListenerObserver`. Mas não podemos mais recebê-lo injetado, pois estamos instanciando manualmente o `PhaseListenerObserver`.

```
@Inject
private Event<PhaseEvent> observer;
private Annotation momento;
```

Mas `Event` é uma interface. Precisamos de alguém que implemente a interface, para que seja possível instanciar um objeto do tipo `Event`. O que fazer? Além da forma que utilizamos até o momento para disparar eventos, também é possível disparar eventos de uma outra forma, utilizando um objeto do CDI.

Falamos desse objeto anteriormente, quando estávamos configurando o CDI. Se verificarmos o arquivo `web.xml`, podemos ver que foi declarado um `BeanManager`:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_
    version="3.0">

    <!-- restante do código !>

    <resource-env-ref>
        <resource-env-ref-name>BeanManager</resource-env-ref-name>
        <resource-env-ref-type>
            javax.enterprise.inject.spi.BeanManager
        </resource-env-ref-type>
    </resource-env-ref>
</web-app>
```

Um dos papéis do `BeanManager` é disparar eventos. Com o `BeanManager` também é possível instanciar objetos programaticamente, o que é conhecido como *loockup*.

No `PhaseListenerObserver`, vamos remover a anotação `@Inject` e substituir o `Event` por um `BeanManager`. Para obter um `BeanManager`, utilizamos a classe `CDI`. É feita, então, uma chamada ao método `current()`, que nos dá acesso ao provedor utilizado, o Weld. Em seguida, utiliza-se o método `getBeanManager()`, para pegar o `BeanManager` que o provedor tem:

```
public class PhaseListenerObserver {

    private BeanManager observer = CDI.current().getBeanManager();
    private Annotation momento;

    // restante do código
}
```

O método `fire()` para de funcionar, porque o `observer` não tem mais o método `select()`. Para disparar um evento com o `BeanManager`, utilizamos o método `fireEvent()`. Passamos dois argumentos para o método: o evento que desejamos disparar e os qualificadores que vamos definir para o evento:

```
public void fire(PhaseEvent phaseEvent) {
    observer.fireEvent(phaseEvent, momento, new PhaseLiteral(phaseEvent.getPhaseId()));
}
```

Agora que realizamos as alterações, vamos instalar a *lib* no repositório local e ver se tudo funciona. Vamos atualizar o projeto `livraria`, dar um "Clean" e iniciar o Tomcat.

No `LogPhase`, não definimos uma fase específica, então após o login, podemos ver no console que temos todas as fases:

```
FASE: RENDER_RESPONSE 6
FASE: RESTORE_VIEW 1
/login.xhtml
FASE: APPLY_REQUEST_VALUES 2
FASE: PROCESS_VALIDATIONS 3
FASE: UPDATE_MODEL_VALUES 4
fazendo login do usuario admin@caelum.com.br
Hibernate: select usuario0_.id as id1_, usuario0_.email as email1_, usuario0_.senha as senha1_ from
FASE: INVOKE_APPLICATION 5
FASE: RESTORE_VIEW 1
/livro.xhtml
FASE: APPLY_REQUEST_VALUES 2
FASE: PROCESS_VALIDATIONS 3


FASE: UPDATE_MODEL_VALUES 4
FASE: INVOKE_APPLICATION 5
```

Temos no projeto `livraria`, o `Autorizador`, que também é um `phaseListener`. Mas não precisamos mais de um `PhaseListener`. É só ter um método e indicar que queremos observar uma determinada fase.

O primeiro passo é remover o `implements PhaseListener` da classe. No método `afterPhase()`, removemos o `@Override`. Por fim, os métodos `beforePhase()` e `getPhaseId()` não são mais necessários:

```
public class Autorizador {

    public void afterPhase(PhaseEvent evento) {

        FacesContext context = evento.getFacesContext();
        String nomePagina = context.getViewRoot().getViewId();
```

```
        System.out.println(nomePagina);

        if("/login.xhtml".equals(nomePagina)) {
            return;
        }

        Usuario usuarioLogado = (Usuario) context.getExternalContext().getSessionMap().get("usuarioL

        if(usuarioLogado != null) {
            return;
        }


        //redirecionamento para login.xhtml

        NavigationHandler handler = context.getApplication().getNavigationHandler();
        handler.handleNavigation(context, null, "/login?faces-redirect=true");
        context.renderResponse();
    }
}
```

Na assinatura do método, vamos adicionar o `@Observes`, e utilizar os qualificadores. Só temos interesse na fase `RESTORE_VIEW`:

```
public void afterPhase(@Observes @After @Phase(Phases.RESTORE_VIEW) PhaseEvent evento) {
```

Temos que remover também do `faces-config.xml`. Para isso, basta comentar a linha que continha o `Autorizador`:

```
<lifecycle>
        <!-- <phase-listener>br.com.alura.livraria.util.Autorizador</phase-listener> -->
        <phase-listener>br.com.alura.alura_lib.jsf.phaselistener.PhaseListenerGenerico</phase-listen
</lifecycle>
```

Ao reiniciar o servidor, tudo funciona. Ainda temos um ponto para melhorar no método `afterPhase()` do `Autorizador`. Ele ainda está obtendo o contexto e não está utilizando nada do que somos capazes de injetar. Vamos adicionar o contexto e o mapa de sessão como dependências:

```
@Inject
private FacesContext context;

@Inject @ScopeMap(Scope.SESSION)
private Map<String, Object> sessionMap;
```

No `afterPhase()`, não precisamos mais do contexto, então podemos remover a seguinte linha:

```
FacesContext context = evento.getFacesContext();
```

Na linha que obtemos o mapa de sessão:

```
Usuario usuarioLogado = (Usuario) context.getExternalContext().getSessionMap().get("usuarioLogado");
```

Vamos alterar para:

```
Usuario usuarioLogado = (Usuario) sessionMap.get("usuarioLogado");
```

Reiniciaremos para ver se as alterações surtiram efeito e veremos que o servidor funciona. Estamos recebendo as dependências injetadas.

Vamos continuar melhorando o método. Na seguinte linha:

```
NavigationHandler handler = context.getApplication().getNavigationHandler();
```

Perceba que está sendo pego o contexto da aplicação para em seguida obter o `NavigationHandler`. Podemos produzir esse objeto. Na biblioteca, na classe `JSFFactory`, do pacote pacote `br.com.alura.alura_lib.factory`, vamos adicionar um método que produz um `NavigationHandler`:

```
public class JSFFactory {

    // outros métodos

    @Produces
    @RequestScoped
    public NavigationHandler getNavigationHandler() {
        return getFacesContext().getApplication().getNavigationHandler();
    }

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

Agora que temos o `NavigationHandler`, podemos injetá-lo no `Autorizador`:

```
@Inject
private NavigationHandler handler;
```

A seguinte linha do método `afterPhase` pode ser excluída:

```
NavigationHandler handler = context.getApplication().getNavigationHandler();
```

O método final ficou mais simples:

```
public void afterPhase(@Observes @After @Phase(Phases.RESTORE_VIEW) PhaseEvent evento) {

    String nomePagina = context.getViewRoot().getViewId();

    System.out.println(nomePagina);

    if("/login.xhtml".equals(nomePagina)) {
        return;
    }

    Usuario usuarioLogado = (Usuario) sessionMap.get("usuarioLogado");

    if(usuarioLogado != null) {
        return;
    }

    //redirecionamento para login.xhtml

    handler.handleNavigation(context, null, "/login?faces-redirect=true");
    context.renderResponse();
}
```

Já que fizemos alterações no `alura-lib` , precisamos fazer todo o processo que já estamos acostumados: instalar a *lib* no repositório local, atualizar o projeto livraria ("Alt + F5"), fazer um "Clean" e iniciar o Tomcat.

Ao testarmos a aplicação, o login continua funcionando! Agora recebemos as nossas dependências injetadas e temos um método que observa o evento.