02

Autorização de usuários com Interceptors

Autorizando o acesso as lógicas com Interceptors

Estamos a um passo de completar essa funcionalidade de login. Repare que da forma que está, mesmo quando o usuário não está logado ele consegue acessar todas as funcionalidades do nosso sistema. Isso não pode continuar assim, apenas quem tem autorização de acesso deveria conseguir acessar as informações e fazer modificações em nosso sistema.

Mas como adicionar esse comportamento em todas as lógicas? Eu até poderia colocar um `if` em todos os meus métodos do `controller`, perguntando para a classe `UsuarioLogado` se ela possui um usuário diferente de `null`, algo como:

```
@Get
public void lista() {
    if (usuarioLogado.getUsuario() != null) {
        result.include("produtoList", dao.lista());
        return;
    }
    result.redirectTo(LoginController.class).formulario();
}
```

Mas imagine o trabalho que isso daria! Além de que essa mudança vai poluir bastante o nosso código, e sempre precisaríamos lembrar de adicionar essa condição ao adicionarmos um novo método.

Essa é uma necessidade muito comum no desenvolvimento web, conseguir executar alguma lógica **durante**, **antes** ou **depois** do código de nossos controllers serem executados. Chamamos uma classe com essa habilidade de `Interceptor`.

Para criar um **interceptor** do VRaptor 4, tudo que precisamos fazer é criar uma classe comum e adicionar a anotação `@Intercepts` nessa classe. Vamos criar um interceptor chamado `AutorizadorInterceptor` dentro do pacote `br.com.caelum.vraptor.interceptor`. Esse interceptor vai fazer essa condição antes de executar o código do nosso `Controller`, e permitir a execução da lógica do nosso `Controller` apenas caso o usuário esteja logado.

```
@Intercepts
public class AutorizadorInterceptor {

}
```

Precisamos adicionar um método com o código que será executado **antes** e **depois** da lógica da nossa aplicação ser executada. Vamos chamá-lo de `intercepta`. Existem algumas regras para esse método funcionar, ou seja, interceptar uma requisição, e a primeira delas é que ele precisa estar anotado com `@AroundCall`.

Outra regra é que esse método precisa receber como parâmetro a classe `SimpleInterceptorStack`, cujo método `next()` vai indicar o ponto em que o código será executado:

```
@Intercepts
public class AutorizadorInterceptor {
```

```
    @AroundCall
    public void intercepta(SimpleInterceptorStack stack){

        // antes de executar meu código

        stack.next(); // executa o código

        // depois de executar meu código
    }
}
```

Excelente, agora só precisamos adicionar a condição para permitir ou não a execução desse método. Assim como nos `Controllers` podemos utilizar **Injeção de Dependências** em nossos `Interceptors` do VRaptor. Vamos então pedir o `UsuarioLogado` injetado, e fazer a condição redirecionando para o `formulario` sem executar nossas lógicas caso o usuário seja nulo.

```
@Intercepts
public class AutorizadorInterceptor {

    private final Result result;
    private final UsuarioLogado usuarioLogado;

    @Inject
    public AutorizadorInterceptor(Result result,
            UsuarioLogado usuarioLogado) {
        this.result = result;
        this.usuarioLogado = usuarioLogado;
    }

    @Deprecated
    AutorizadorInterceptor() {
        this(null, null); // para uso do CDI
    }

    @AroundCall
    public void intercept(SimpleInterceptorStack stack) {

        if (usuarioLogado.getUsuario() == null) {
            result.redirectTo(LoginController.class).formulario();
            return;
        }
        stack.next();
    }
}
```

Agora que tudo parece estar pronto, vamos tentar acessar a página de login da nossa aplicação: http://localhost:8080/vraptor-produtos/login/formulario (http://localhost:8080/vraptor-produtos/login/formulario). Repare que após tentar carregar a página por um tempo, o navegador nos respondeu com a seguinte mensagem: *"This webpage has a redirect loop"*.

Isso aconteceu devido a condição que adicionamos em nosso interceptor! Repare:

```
if (usuarioLogado.getUsuario() == null) {
    result.redirectTo(LoginController.class).formulario();
    return;
}
```

Para acessar qualquer lógica (inclusive a do método `formulario` que é apenas redirecionar para sua `jsp`) a condição `usuarioLogado.getUsuario() == null` vai ser sempre **true**. Afinal, sem abrir o formulário e enviar os dados para o método `autentica` o objeto `usuarioLogado` nunca vai ter um usuário válido.

Determinando quais métodos serão interceptados

Precisamos ensinar ao interceptor do VRaptor que esses métodos não devem ser interceptados! Para fazer isso, basta adicionar um método anotado com `@Accepts` em nosso interceptor. Esse método deve retornar um **boolean** que é a condição que a classe precisa atender para ser interceptada:

```
@Accepts
public boolean accepts() {
    return // alguma condição aqui
}
```

Um caso de uso muito comum do `accepts` é quando marcamos um método de nosso projeto com alguma anotação nossa, assim dentro do `accepts` só precisamos validar se o método que está sendo interceptado possui essa anotação, assim definir se vamos ou não interceptar aquele método. Por exemplo, vamos criar a anotação `@Public` dentro do pacote `br.com.caelum.vraptor.annotation`:

```
@Documented
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.METHOD)
public @interface Public {
}
```

Agora, no método `formulario` e `autentica` da classe `LoginController` adicionamos essa nova `annotation`:

```
@Public
@Get
public void formulario() {
}

@Public
@Post
public void autentica(Usuario usuario) {
    // código omitido
}
```

E em nosso método `accepts`, do `AutorizadorInterceptor` dizemos que só deve aceitar (*interceptar*) se o método possuir essa anotação que criamos, a `@Public`. Mas como saber qual o método que estamos interceptando? Isso é bem simples, o VRaptor possui uma objeto em cada requisição que guarda essa informação, o `ControllerMethod`. Vamos pedir essa classe injetada em nosso Interceptor:

```
@Intercepts
public class AutorizadorInterceptor {

    private final Result result;
    private final UsuarioLogado usuarioLogado;
    private final ControllerMethod controllerMethod;

    @Inject
    public AutorizadorInterceptor(Result result,
            UsuarioLogado usuarioLogado, ControllerMethod controllerMethod) {
        this.result = result;
        this.usuarioLogado = usuarioLogado;
        this.controllerMethod = controllerMethod;
    }

    // restante do código omitido
}
```

Agora só precisamos adicionar a condição do método `accepts`. Ele só deve interceptar se o método não contiver a anotação `@Public`, portanto:

```
@Accepts
public boolean accepts() {
    return !controllerMethod.containsAnnotation(Public.class);
}
```

Para testar, vamos restartar o servidor e acessar a listagem de produtos: http://localhost:8080/vraptor-produtos/produto/lista (http://localhost:8080/vraptor-produtos/produto/lista). Repare que agora, como não estamos logados, fomos redirecionados para a página de login com sucesso!

Vimos que para o caso do controle de acesso, do nosso `AutorizadorInterceptor`, foi muito conveniente utilizar o método anotado com `@AroundCall`, pois assim recebemos o `SimpleInterceptorStack` como parâmetro e assim é possível escolher se vamos ou não chamar o método `next` que permite a continuação do nosso código e assim a execução da lógica dos nossos `controller`s.

Interceptando antes ou depois da execução

Mas note que existem algumas situações em que não queremos interceptar durante a execução de nosso lógica, e sim apenas antes, ou até mesmo apenas depois que tudo aconteceu. E nem sempre queremos condicionar se vamos ou não executar a lógica. Isso pode ser feito com o `@AroundCall`, claro. Eu poderia simplesmente inserir o código que quero executar antes da chamada do método `next`, ou apenas depois.

Mas para simplificar ainda mais esse trabalho, o VRaptor conta com mais dois tipos de métodos nesse novo modelo de interceptor, o `@AfterCall` e o `@BeforeCall`. Vejamos um exemplo:

```
@Intercepts
public class AlgumInterceptor {

    @BeforeCall
    public void before() {
        // código a ser executado antes da lógica
```

```
    }

    @AfterCall
    public void after() {
        // código a ser executado depois da lógica
    }
}
```

Repare que para estes casos nós não precisamos injetar o `SimpleInterceptorStack`. A chamada do método `next` é implicita!

Vamos agora criar um `Interceptor` que nos mostra um log simples, com o nome do método que está sendo executado.

Bem, criar o interceptor nós já vimos como fazer, assim como um método `@BeforeCall`. Vamos então começar criando a classe `LogInterceptor` no pacote `br.com.caelum.vraptor.interceptor` e adicionar esse método:

```
@Intercepts
public class LogInterceptor {

    @BeforeCall
    public void before() {
        // como saber qual método está sendo executado?
    }
}
```

Assim como fizemos com o `AutorizadorInterceptor`, podemos pedir injetado o `ControllerMethod` para obter o nome do método que será executado. Vamos adicionar o contrutor e atributo no `LogInterceptor`:

```
private final ControllerMethod controllerMethod;

@Inject
public LogInterceptor(ControllerMethod controllerMethod) {
    this.controllerMethod = controllerMethod;
}
```

Agora só precisamos *printar* o nome do método que está sendo executado! Algo como:

```
@Intercepts
public class LogInterceptor {

    private final ControllerMethod controllerMethod;

    @Inject
    public LogInterceptor(ControllerMethod controllerMethod) {
        this.controllerMethod = controllerMethod;
    }

    public LogInterceptor() {
        this(null); // para uso do CDI
    }

    @BeforeCall
```

```
    public void before() {
        String nomeDoMetodo = controllerMethod.getMethod().getName();
        System.out.println("Executando o método: " + nomeDoMetodo);
    }
}
```

Fazendo o mesmo teste que fizemos anteriormente, restartando o tomcat e acessando nossa listagem de produtos, repare que no console temos a seguinte saída:

```
Executando o método: lista
```

Interceptando só os métodos anotados

Não queremos mostrar o log com o nome de todos os métodos executados, pois isso vai deixar nosso `console` cheio de mensagens a todo momento. Em quero controlar quais métodos serão logados. Já vimos como fazer isso, podemos criar uma anotação para marcar esses métodos!

Vamos criar a anotação `@Log` dentro do pacote `br.com.caelum.vraptor.annotation` :

```
@Documented
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.METHOD)
public @interface Log {
}
```

Agora, no método `formulario` do nosso `ProdutoController` adicionamos essa nova `annotation` :

```
@Log
@Get
public void formulario() {
}
```

E em nosso método `accepts` , do `LogInterceptor` dizemos que só deve aceitar (interceptar) se o método possuir essa anotação que criamos, a `@Log` :

```
@Accepts
public boolean accepts() {
    return method.containsAnnotation(Log.class);
}
```

Para testar, basta restartar o servidor e acessar a página de lista de produtos: http://localhost:8080/vraptor-produtos/produto/lista (http://localhost:8080/vraptor-produtos/produto/lista). Repare que nenhuma informação foi logada, afinal o método `lista` não possui a anotação `@Log` .

Vamos acessar agora o nosso formulário: http://localhost:8080/vraptor-produtos/produto/formulario (http://localhost:8080/vraptor-produtos/produto/formulario). Veja que a saída do console foi:

```
Executando o método: formulario
```

Trabalhando com os accepts customizados

Para os casos mais comuns (como o que fizemos agora), o VRaptor nos provê alguns `accepts` customizados! Por exemplo, para ter o mesmo efeito que nosso método `accepts` do `LogInterceptor`, poderíamos ter anotado essa classe com `@AcceptsWithAnnotations`, dessa forma:

```
@AcceptsWithAnnotations(Log.class)
@Intercepts
public class LogInterceptor {
    // continuação do código omitida
```

Vamos fazer um teste, basta apagar o nosso método `accepts` e manter apenas esse `custom accepts`.

O outro `accept` customizado é o `@AcceptsWithPackage`, que como o proprio nome indica, só intercepta as classes de um determinado pacote.

Controlando a transação do JPA com interceptor

Agora que conhecemos o essencial de `Interceptor`s, podemos usá-lo para melhorar um ponto de nosso código. Repare que nos métodos `adiciona` e `remove` da classe `ProdutoDao` foi preciso iniciar uma transação antes de executar a operação, e logo em seguida `commit`ar essa transação, para que tudo seja realmente aplicado.

É essencial trabalhar com transações quando estamos trabalhando com banco de dados, afinal precisamos garantir a integridade de nossa base de dados, só modificar um valor caso tudo realmente de certo em nosso código.

É bastante trabalhoso fazer esse controlle de transação manualmente, sempre que vamos aplicar uma operação que modifique algum dado no banco precisamos lembrar de abrir e `commit`ar a transação, além de que o código de todos os nossos `DAO`s ficam bastante acoplados com a forma de controlar transações do framework que estivermos usando.

Podemos automatizar esse processo criando um `Interceptor` que inicie uma transação quando uma requisição acontecer, e termine(`commit`) essa transação antes que a resposta seja enviada ao cliente! Vamos chamá-lo de `ControleDeTransacaoInterceptor`:

```
@Intercepts
public class ControleDeTransacaoInterceptor {

    private final EntityManager em;

    @Inject
    public ControleDeTransacaoInterceptor(EntityManager em) {
        this.em = em;
    }

    @Deprecated
    ControleDeTransacaoInterceptor() {
        this(null); // para uso do CDI
    }

    @AroundCall
```

```
    public void intercept(SimpleInterceptorStack stack) {
        em.getTransaction().begin();
        stack.next();
        em.getTransaction().commit();
    }
}
```

Agora podemos **apagar as linhas** de transação do nosso `ProdutoDao`, desamarrando ele dessa responsabilidade.

Bem simples, não acha? Essa é uma prática muito comum do dia a dia de quem trabalha com JPA.

Um ponto importante é que mesmo que não use estamos abrindo transação em todas as requests! Poderíamos evitar isso criando uma anotação pra marcar quais métodos queremos que possuam uma transação, assim como fizemos com o `@Log`. Mas o custo de abrir uma transação e não utilizar não é grande, por isso acaba virando uma prática comum abrir essa transação sempre e não ter a obrigação de anotar todos os métodos que precisar dela, além de que evitamos o risco de esquecer dessa anotação em um método que precisaria da transação.

Ordenando a execução dos interceptors, after e before

Repare que em nenhum momento configuramos a ordem que queremos executar nossos interceptor. Em alguns casos isso é extremamente importante! Por exemplo, o interceptor `AutorizadorInterceptor` deveria ser o primeiro a ser executado, afinal se o usuário não for autorizado não precisamos abrir uma transação ou logar as informações do método que ele está tentando acessar.

Podemos definir essa ordem utilizando os atributos **after** e **before** da anotação `@Intercepts`. Por exemplo, para dizer que o `ControleDeTransacaoInterceptor` deve ser executados depois do `AutorizadorInterceptor`, podemos adicionar o atributo `after=AutorizadorInterceptor.class`:

```
@AcceptsWithAnnotations(Log.class)
@Intercepts(after=AutorizadorInterceptor.class)
public class ControleDeTransacaoInterceptor {
```

Também podemos receber mais de uma classe nos atributos `after` e `before`, por exemplo, agora o `ControleDeTransacaoInterceptor` deve ser executado depois do `AutorizadorInterceptor` e `LogInterceptor`:

```
@AcceptsWithAnnotations(Log.class)
@Intercepts(after={
    AutorizadorInterceptor.class,
    LogInterceptor.class
})
public class ControleDeTransacaoInterceptor {
```

Para testar, vamos adicionar a anotação `@Log` no método `lista` do `ProdutoController`. Agora basta reiniciar o servidor e tentar acessar esse método antes de fazer o login na aplicação. Repare que o log do método `lista` não apareceu no console. Vamos fazer o login e acessar novamente o método `lista`. Agora sim o log está disponível no console.