05

O primeiro Controller

Transcrição

Criaremos agora nosso primeiro *controller*! O *controller* é nada mais nada menos que uma classe Java, então crie uma classe com o nome de `HomeController` no pacote `br.com.casadocodigo.loja.controllers`.

```
package br.com.casadocodigo.loja.controllers;

public class HomeController{

}
```

O que temos aqui não é nada mais que uma simples classe java. Nós ainda não dissemos para o SpringMVC que esta classe é um *controller*! Podemos fazer esta definição de duas formas: 1. Usando regras de XML. 2. Usando anotaçoes Java.

Por questões de simplicidade, usaremos a segunda forma, para isto só precisamos adicionar a anotação `@Controller` logo acima da definição da nossa classe:

```
package br.com.casadocodigo.loja.controllers;

@Controller
public class HomeController{

}
```

Note que apesar de termos feito todos estes passos, o Eclipse marca de vermelho a anotação `@Controller`, como se ele não entendesse a anotação. Isso acontece porque em momento nenhum configuramos o SpringMVC para nosso projeto. Para isso precisamos declarar algumas dependências no arquivo `pom.xml` para que o Maven baixe e configure o SpringMVC para o nosso projeto.

Abra o arquivo `pom.xml` e na aba `pom.xml` cole as dependências do projeto logo abaixo do fechamento da tag `<properties>`.

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.1.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-servlet-api</artifactId>
        <version>7.0.30</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
```

```
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp.jstl</groupId>
            <artifactId>jstl-api</artifactId>
            <version>1.2</version>
            <exclusions>
                <exclusion>
                    <groupId>javax.servlet</groupId>
                    <artifactId>servlet-api</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.glassfish.web</groupId>
            <artifactId>jstl-impl</artifactId>
            <version>1.2</version>
            <exclusions>
                <exclusion>
                    <groupId>javax.servlet</groupId>
                    <artifactId>servlet-api</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>1.6.1</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.6.1</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

Salve o `pom.xml` e aguarde um momento, o Maven deverá baixar e configurar as dependências do nosso projeto. Um novo diretório chamado `Maven Dependencies` deve aparecer no projeto (Isso pode demorar um pouco já que o Maven irá baixar as dependências).

Tudo parece funcionar perfeitamente! Vamos agora criar um metodo em nossa classe que será responsável por atender as requisições que chegam na página inicial do nosso projeto, ou seja, o endereço raiz ("/"). Em nosso controller teremos o código:**Observação:** Lembre-se de fazer os imports necessários (Atalho: Ctrl + Shift + O)

```
[...]
@Controller
public class HomeController{

    @RequestMapping("/")
    public void index(){
        System.out.println("Entrando na home da CDC");
    }

}
```

Uma novidade em nosso código é a anotação `@RequestMapping` , que tem a função de definir que aquele metodo atende a um determinado `path` ou endereço. Neste caso estamos definindo que o metodo `index` atenderá as requisições que chegarem na raiz do nosso projeto `("/")` .Se entrarmos em nosso projeto agora, deveriamos visualizar a mensagem do escrita no `System.out.println` certo? Ainda não!

Em nenhum momento até agora, configuramos o SpringMVC para atender as requisições que chegam para nossa aplicação. Se acessarmos através do navegador o nosso projeto, veremos a mensagem de `Oi` que fizemos no arquivo `index.html` .

Existem duas formas de configurarmos o servidor Tomcat para que ele passe as requisições para o SpringMVC: 1. Usando o Servets 2. Usando os Filtros

Usaremos a primeira opção, pois o SpringMVC já vem com um `servlet` pronto para utilizarmos como `servlet` de configuração. Podemos configura-lo através de XML ou código Java e mais uma vez usaremos código java!

Crie uma nova classe dentro do pacote `br.com.casadocodigo.loja.conf` chamada de `ServletSpringMVC` . Com a classe criada, faça ela extender a classe: **AbstractAnnotationConfigDispatcherServletInitializer**.

```
public class ServletSpringMVC extends AbstractAnnotationConfigDispatcherServletInitializer{

}
```

A classe `ServletSpringMVC` parece estar com erros agora que extendemos a classe do SpringMVC. A classe do SpringMVC contém alguns métodos que precisamos implementar! Clique sobre o nome da Classe `ServletSpringMVC` use o atalho `Ctrl + 1` e selecione a opção: `Add Unimplemented Methods` . Veja que o Eclipse completou nossa classe com alguns metodos:

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return null;
}

@Override
protected Class<?>[] getServletConfigClasses() {
    return null;
}
```

```
@Override
protected String[] getServletMappings() {
    return null;
}
```

Implementaremos os dois últimos metodos. O **getServletConfigClasses** pede um `array` de classes de configurações, que ainda não temos, mas que iremos criar depois. O **getServletMappings** pede um `array` com os mapeamentos que queremos que nosso servlet atenda. Nossa implementação ficará assim:

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[] {AppWebConfiguration.class};
}

@Override
protected String[] getServletMappings() {
    return new String[] {"/"};
}
```

Como estas configurações estamos definindo que o `servlet` do SpringMVC atenderá as requisições a partir da raiz do nosso projeto `(/)` e que a classe `AppWebConfiguration` será usada como classe de configuração do servlet do SpringMVC.Crie a classe `AppWebConfiguration` no pacote `br.com.casadocodigo.loja.conf` .

Na classe `AppWebConfiguration` nós precisamos usar o recurso de Web MVC do SpringMVC. Podemos fazer isso através de mais uma anotação. Antes da declaração da classe devemos adicionar a anotação: `@EnableWebMvc` .

```
@EnableWebMvc
public class AppWebConfiguration {

}
```

Precisamos também configurar o caminho (pacote) onde o SpringMVC irá encontrar os nossos *controllers*! Mais uma anotação para esta configuração é necessária: `@ComponentScan` .Isso somente não resolve nosso problema, o **ComponentScan** precisa saber onde procurar os nossos `controllers` e podemos indicar isso passando um paramêtro. Exitem duas possibilidades:1. `@ComponentScan(basePackages={""})` que recebe um array de **Strings** com os nomes dos pacotes onde o SpringMVC pode encontrar os `controllers` .

1. `@ComponentScan(basePackageClasses={})` que recebe um array de **classes** de onde o SpringMVC pode extrair os pacotes nos quais ele pode encontrar os `controllers` .

A segunda opção é a melhor escolha, pois podemos passar classes que estão no mesmo pacote dos `controllers` e assim o SpringMVC descobre o pacote de forma automática, diferente da primeira opção, que é mais manual.Com a primeira opção, caso o pacote mude, temos que lembrar de modificar essa configuração também.Para a segunda opção, passaremos então nossa classe `HomeController` . Assim o SpringMVC descobre o pacote desta classe e pode encontrar os outros controllers automáticamente.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class})
public class AppWebConfiguration {
```

```
}
```

Falta testar o nosso primeiro *controller*, mas isso fica para o próximo vídeo.