### 目录

#### 第一章 掌握要点

1.深入理解Spring Security及相关框架的原理、功能和代码

2.可以基于Spring Security及相关框架独立开发认证授权相关功能

3.掌握抽象和封装的常见技巧，可以编写可重用的模块供他人使用

#### 第二章 开始开发

1.环境安装

2.架构介绍

3.Hello Spring Security

#### 第三章 使用Spring MVC开发RESTful API

1.开发基本增删改查接口

2.Spring MVC高级特性

3.REST服务开发常用工具

#### 第四章 Spring Security开发基于表单的认证

1.Spring Security原理介绍

2.基于Spring Security的默认实现开发“用户名+密码”认证

3.开发“手机号+短信”认证

#### 第五章 Spring Social开发第三方认证

1.OAuth协议 以及Spring Social原理简介

2.实现QQ认证和微信认证

3.SESSION管理及退出

#### 第六章 Spring Security OAuth开发APP认证模块

1.Spring Security Oauth简介

2.重构原有认证功能，使其支持基于Token的认证方式

3.基于OAuth协议实现SSO（单点登录）

#### 第七章 Spring Security授权控制

1.Spring Security授权机制介绍

2.实现3种不同复杂度的授权机制

3.重构代码

### 第二章 开始开发

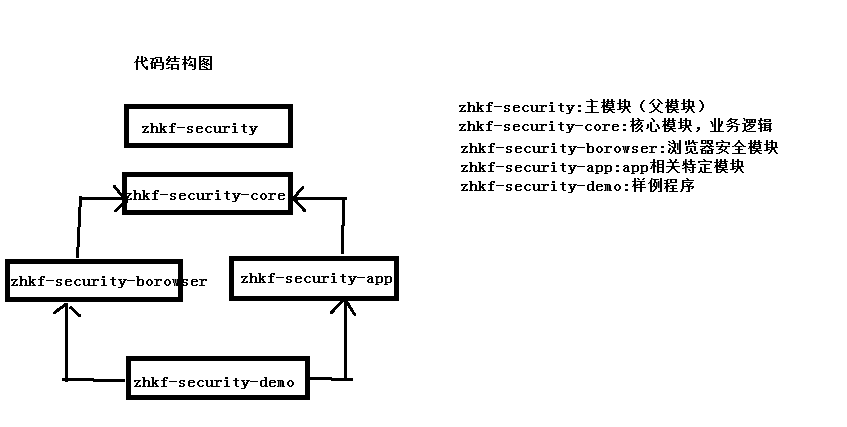
#### 一、开发环境安装

1.安装JDK

2.安装STS(<https://spring.io/tools/sts)/idea/Eclipse>.(这里我使用的是Idea)

3.安装MySql/Redis

#### 二、代码结构



#### 三、hello Spring Security

##### 1.创建项目模块

利用编辑工具或官方网站生成spring项目，创建方式可参见<https://blog.csdn.net/qq_43251032/article/details/100895782>。

创建成功后，各个模块的pom文件依赖为：

zhkf-security:

<**modelVersion**>4.0.0</**modelVersion**>  
<**groupId**>com.zhkf.security</**groupId**>  
<**artifactId**>zhkf-security</**artifactId**>  
<**version**>1.0.0-SNAPSHOT</**version**>  
<**packaging**>pom</**packaging**>

<**modules**>  
 <**module**>../zhkf-security-core</**module**>  
 <**module**>../zhkf-security-borowser</**module**>  
 <**module**>../zhkf-security-app</**module**>  
 <**module**>../zhkf-security-demo</**module**>  
</**modules**>

zhkf-security-core:

<**modelVersion**>4.0.0</**modelVersion**>  
<**artifactId**>zhkf-security-core</**artifactId**>  
<**parent**>  
 <**groupId**>com.zhkf.security</**groupId**>  
 <**artifactId**>zhkf-security</**artifactId**>  
 <**version**>1.0.0-SNAPSHOT</**version**>  
 <**relativePath**>../zhkf-security</**relativePath**>  
</**parent**>

其它子模块与zhkf-security-core的依赖配置相似。

##### 2.配置模块

（1）zhkf-security:

①声明版本版本变量

<**properties**>  
 <**zhkf.security.version**>1.0.0-SNAPSHOT</**zhkf.security.version**>  
</**properties**>

②依赖版本管理插件

<**dependencyManagement**>  
 <**dependencies**>  
 <!--管理Maven依赖的版本,以下两个插件可以在引入其它插件时，不用单独写版本-->  
 <**dependency**>  
 <**groupId**>io.spring.platform</**groupId**>  
 <**artifactId**>platform-bom</**artifactId**>  
 <**version**>Brussels-SR4</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 <!--Spring Cloud依赖-->  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Dalston.SR2</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

③环境版本管理

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>2.3.2</**version**>  
 <**configuration**>  
 <**source**>1.8</**source**>  
 <**target**>1.8</**target**>  
 <**encoding**>UTF-8</**encoding**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

(2).zhkf-core:

①oauth2、第三方登录插件必用的

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-oauth2</**artifactId**>  
</**dependency**>

②redis依赖(加入该依赖，电脑必须运行安装的redis环境)

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
 </**dependency**>

③JDBC存储依赖

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-jdbc</**artifactId**>  
</**dependency**>

④mySql依赖

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
</**dependency**>

⑤SpringSocial,用于三方登录的三大插件

<**dependency**>  
 <**groupId**>org.springframework.social</**groupId**>  
 <**artifactId**>spring-social-config</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.social</**groupId**>  
 <**artifactId**>spring-social-core</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.social</**groupId**>  
 <**artifactId**>spring-social-security</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.social</**groupId**>  
 <**artifactId**>spring-social-web</**artifactId**>  
</**dependency**>

⑥Java常用工具包插件

<**dependency**>  
 <**groupId**>commons-lang</**groupId**>  
 <**artifactId**>commons-lang</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>commons-collections</**groupId**>  
 <**artifactId**>commons-collections</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>commons-beanutils</**groupId**>  
 <**artifactId**>commons-beanutils</**artifactId**>  
</**dependency**>

1. zhkf-security-borowser:

由于zhkf-security-core引入了比较全面的插件，这里我们只需将zhkf-security-core引入过来即可。

<**dependency**>  
 <**groupId**>com.zhkf.security</**groupId**>  
 <**artifactId**>zhkf-security-core</**artifactId**>  
 <**version**>${zhkf.security.version}</**version**>  
</**dependency**>

另外会用到一个Session插件，在这里也安装一下

<**dependency**>  
 <**groupId**>org.springframework.session</**groupId**>  
 <**artifactId**>spring-session</**artifactId**>  
</**dependency**>

1. zhkf-security-app

这个后面在会讲到，这里先把zhkf-security-core的插件引入过来

<**dependency**>  
 <**groupId**>com.zhkf.security</**groupId**>  
 <**artifactId**>zhkf-security-core</**artifactId**>  
 <**version**>${zhkf.security.version}</**version**>  
</**dependency**>

1. zhkf-security-demo

先加入springboot依赖与zhkf-security-borowser相关依赖

<**dependency**>  
 <**groupId**>com.zhkf.security</**groupId**>  
 <**artifactId**>zhkf-security-borowser</**artifactId**>  
 <**version**>${zhkf.security.version}</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
</**dependency**>

依赖先加到此处，可以发现插件没有任何版本信息，只因为父模块中引入了版本控制相关插件。

##### demo实现

在zhkf-security-demo的java文件夹下，新建文件com.zhkf.DemoApplication.java文件，如果是springboot初始项目，默认会有启动入口类，代码大概如下：

@SpringBootApplication  
**public class DemoApplication** {  
 **public static void** main(**String**[] args){  
 **SpringApplication**.run(**DemoApplication**.**class**,args);  
 }  
}

现将代码改写如下：

@SpringBootApplication  
@RestController  
**public class DemoApplication** {  
 **public static void** main(**String**[] args){  
 **SpringApplication**.run(**DemoApplication**.**class**,args);  
 }  
 @GetMapping("/hello")  
 **public String** hello(){  
 **return** "hello spring security";  
 }  
}

并在resource的额application.properties文件下配置以下内容

#数据库四要素  
**spring.datasource.driver-class-name** = com.mysql.jdbc.Driver  
**spring.datasource.url**=jdbc:mysql://127.0.0.1:3306/zhkf-demo?useUnicode=yes&characterEncoding=UTF-8&useSSL=false  
**spring.datasource.username**=root  
**spring.datasource.password**=yj123456  
#关闭session管理  
**spring.session.store-type**=*none*#关闭SpringSecurity安全管理器  
**security.basic.enabled**=**false**#自定义端口号  
**server.port**=8060

改写完成之后，启动我们的zhkf-security-demo入口程序，在浏览器中输入http://localhost:8060就可以在页面中看到“hello spring security”。

### 第三章 使用Spring MVC开发RESTful API

#### 一、使用Spring MVC编写Restful API

##### 1. 常见的接口请求与Restful的写法区别

查询 /user/query?name=tom GET /user?name=tom GET

详情 /user/getInfo?id=1 GET /user/1 GET

创建 /user/create?name=tom POST /user POST

修改 /user/update?id=1&name=jquery POST /user/1 PUT

删除 /user/delete?id=1 GET /user/1 DELETE

Resultful特点：

1.使用URL描述资源

2.使用HTTP方法描述行为。使用HTTP状态码来表示不同的结果

3.使用json交互数据，传统模式使用的是键值对形式

4.RESTful只是一种风格，并不是强制的标准。

##### 2. 编写一个Restful API

在zhkf-security-demo的test.java文件夹下新建com.zhkf.web.controller.UserControllerTest.java

类。伪造一分mvc环境，并编写一个测试用例

@RunWith(**SpringRunner**.**class**)  
@SpringBootTest  
**public class UserControllerTest** {  
 @Autowired  
 **private** WebApplicationContext wac;  
  
 **private MockMvc** mockMvc;  
 @Before  
 **public void** setup(){  
 mockMvc= **MockMvcBuilders**.webAppContextSetup(wac).build();  
 }  
 @Test  
 **public void** whenQuerySuccess() **throws Exception**{  
 mockMvc.perform(MockMvcRequestBuilders.get("/user") //发送的请求  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8))  
 .andExpect(MockMvcResultMatchers.status().isOk()) //返回的状态码  
 .andExpect(MockMvcResultMatchers.jsonPath("$.length()").value(3)); //返回一个集合，集合的长度是3  
 }  
}

在main.java文件夹下新建com.zhkf.web.controller.UserController.java

##### 3. 常用注解与传递参数

①@RestController标明此Controller提供RestAPI

在前面建立的UserController类上添加@RestController注解。

②@RequestMapping及其变体。映射http请求url到Java方法

在该类中返回一分查询用户的方法集合

@RequestMapping(value="/user",method = RequestMethod.GET)  
**public** List<**User**> getUser(){  
 **return** users;  
}

在main.java文件夹下新建com.zhkf.dto.User.java,填写两个测试属性：

**private String** username;  
 **private String** password;

//get、set方法省略

这时运行测试返回结果会报错，测试类中返回结果的长度为3，controller的查询结果返回是空，所以报错。需要在返回结果里添加三个用户：

**public** List<**User**> getUser(){  
 List<**User**> users=**new** ArrayList<>();  
 users.add(**new** User());  
 users.add(**new** User());  
 users.add(**new** User());  
 **return** users;  
}

这样案例就就建立起来了。

③@RequestParam映射请求参数到Java方法的参数

方法中添加参数

**public** List<**User**> getUser(@RequestParam **String** username){…

运行后，会出现参数为空的错误，要在测试方法中加入传递的参数

**public void** whenQuerySuccess() **throws Exception**{  
 mockMvc.perform(MockMvcRequestBuilders.get("/user") //发送的请求  
 .param("username","zh")

…

这时控制台中就会把zh打印在控制台中。

方法参数的名字是要与传递参数的名字一致的，如果改变了方法参数的名字变成了nickname,那么只需要改变@RequestParam的name属性即可

**public** List<**User**> getUser(@RequestParam(name="username") **String** nickname){

当方法参数没有传递时，也就是@RequestParam的required=false时，defaultValue可以设置默认值。

**public** List<**User**> getUser(@RequestParam(name="username",required = **false**,defaultValue ="tom") **String** nickname){

这样控制台中打印的结果就为tom。

在着，实际开发中会遇到方法参数很多的情况，这里我们就可以把添加对象参数来调用。

步骤，在dto包里新建UserQueryCondition类，添加参数

**private String** username;  
**private int** age;  
**private int** ageTo;  
**private String** xxx;

UserController中查询方法修改：

**public** List<**User**> getUser(**UserQueryCondition** condition){  
 //利用Java的反射查询  
 **System**.out.println(**ReflectionToStringBuilder**.toString(condition, ToStringStyle.MULTI\_LINE\_STYLE));

UserControllerTest.java:

mockMvc.perform(MockMvcRequestBuilders.get("/user") //发送的请求  
 .param("username","zh")  
 .param("age","20")  
 .param("ageTo","60")  
 .param("xxx","yyy")

控制台打印效果：

com.zhkf.dto.UserQueryCondition@13c886f[

username=zh

age=20

ageTo=60

xxx=yyy

]

④PageableDefault指定分页参数默认值

**public** List<**User**> getUser(**UserQueryCondition**

condition,@PageableDefaule(page=2,size=17,sort="username,asc") Pageable pageable){

System.out.println(pageable.getPageSize());

@PageableDefaule是属于springdata包中的注解，使用时需要开启 springboot-start-data-redis这个依赖。

*有关.jsonPath("$.length()")的用法参见*[*https://github.com/json-path/JsonPath*](https://github.com/json-path/JsonPath)*。*

#### 二、编写用户详情服务

##### [1.使用@PathVariable](mailto:1.使用@PathVariable)映射url片段到java方法的参数

UserControllerTest.java测试用例中添加：

@Test  
**public void** whenGenInfoSuccess() **throws Exception**{  
 mockMvc.perform(MockMvcRequestBuilders.get("/user/1")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8))  
 .andExpect(MockMvcResultMatchers.status().isOk())  
 .andExpect(MockMvcResultMatchers.jsonPath("$.username").value("tom"));  
}

UserController.java服务调用中添加：

@RequestMapping(value="/user/{id}",method=RequestMethod.GET)  
**public User** getInfo(@PathVariable **String** id){  
 **User** user = **new** User();  
 user.setUsername("tom");  
 **return** user;  
}

编译运行whenGenInfoSuccess()方法，可以运行成功。

##### 2.在url声明中使用正则表达式

测试先行，编写UserCOntrollerTest.java

@Test  
**public void** whenGetInfoFail() **throws Exception**{  
 mockMvc.perform(MockMvcRequestBuilders.get("/user/a")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8))  
 .andExpect(MockMvcResultMatchers.status().is4xxClientError());  
}

这是传递的参数是‘a’这个字符串，UserController.java接收的id属性是没有定义规则 的，这时编译运行会成功。现在将UserController.java的getInfo()方法只能接收数字，那么 我们将要修改这个方法。

@RequestMapping(value="/user/{id}",method=RequestMethod.GET)

改为

@RequestMapping(value="/user/{id:\\d+}",method=RequestMethod.GET)

##### 3．使用JsonView控制json输出内容

@JsonView使用步骤：

使用接口来声明多个视图

在值对象的方法上指定视图

在Controller方法上指定视图

###### (1).使用接口来声明多个视图

在User.java中声明两个接口

**public interface** UserSimpleView{};  
**public interface** UserDetailView **extends** UserSimpleView{}

###### (2). 在值对象的方法上指定视图

在username与password的get方法上指定视图

@JsonView(UserSimpleView.**class**)  
**public String** getUsername() {  
 **return** username;  
}

@JsonView(UserDetailView.**class**)  
**public String** getPassword() {  
 **return** password;  
}

###### (3). 在Controller方法上指定视图

在UserController中的getUser()与getInfo()方法上分别添加

@JsonView(**User**.UserSimpleView.**class**)

与

@JsonView(**User**.UserDetailView.**class**)

修改测试类的打印方式(将whenGentInfoSuccess()方法修改以下)：

@Test  
**public void** whenGenInfoSuccess() **throws Exception**{  
 **String** result= mockMvc.perform(MockMvcRequestBuilders.get("/user/1")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8))  
 .andExpect(MockMvcResultMatchers.status().isOk())  
 .andExpect(MockMvcResultMatchers.jsonPath("$.username").value("tom"))  
 .andReturn().getResponse().getContentAsString(); //将服务器返回的字符串以json的格式输出  
 **System**.out.println(result);  
}

其它测试方法也可以相同方式修改，在控制台中查看输出结果。

**代码重构：**

将：@RequestMapping(value="/user",method = RequestMethod.GET)

改为：@GetMapping("/user")

其它服务方法注解也如上修改。

在进一步，在UserController类上声明@RequestMapping注解。定义为@RequestMap-

-ping(“/user”),类中各方法的value路径去掉“/user”即可。

#### 三、用户创建请求

##### 1.@RequestBody映射请求体到Java方法的参数

在测试类中编写一个测试成功后的创建场景

@Test  
**public void** whenCreateSuccess() **throws Exception**{  
 **String** content="{\"username\":\"tom\",\"password\":null}";  
 mockMvc.perform(MockMvcRequestBuilders.post("/user")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8)  
 .content(content)).andExpect(MockMvcResultMatchers.status().isOk())  
 .andExpect((MockMvcResultMatchers.jsonPath("$.id").value("1")));  
}

User实体类中添加一个String类型的id，并在get方法上加入@JsonView注解。

UserController添加方法：

@PostMapping  
**public User** create(**User** user){  
 **System**.out.println(user.getId());  
 **System**.out.println(user.getUsername());  
 **System**.out.println(user.getPassword());  
 user.setId("1");  
 **return** user;  
}

运行测试类后，会返回一个三个null值，原因是未加入@RequestBody注解，修改方法为：

**public User** create(@RequestBody **User** user)就会运行成功

##### 2.日期类型参数的处理

User实体类中添加java.util.Date类型的brityday字段属性，并在get方法上加入@JsonView注解。修改测试类的whenCreateSuccess()方法：

@Test  
**public void** whenCreateSuccess() **throws Exception**{  
 **Date** date=**new** Date();  
 **System**.out.println(date.getTime());  
 **String** content="{\"username\":\"tom\",\"password\":null,\"brityday\":"+date.getTime()+"}";  
 **String** result= mockMvc.perform(MockMvcRequestBuilders.post("/user")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8)  
 .content(content)).andExpect(MockMvcResultMatchers.status().isOk())  
 .andExpect((MockMvcResultMatchers.jsonPath("$.id").value("1")))  
 .andReturn().getResponse().getContentAsString();  
 **System**.out.println(result);  
}

##### 3.Valid注解和BindingResult验证请求参数的合法性并处理校验结果

在User实体类的password属性上加上字符不能为空注解@NotBlank,但是测试类设置的密码是空的，运行方法后，发现并没有报错。这种情况是应为服务方法中没有添加@Valid校验注解。

**public User** create(@Valid @RequestBody **User** user){

当请求失败时，服务方法就会被终结掉，进入不了，有时我们实际开发中需要进入该方法中，

这时@BindingResult注解就起到了作用，修改服务方法：

@PostMapping  
**public User** create(@Valid @RequestBody **User** user, BindingResult errors){  
  
 **if**(errors.hasErrors()){  
 errors.getAllErrors().stream().forEach(error ->**System**.out.println(error.getDefaultMessage()));  
 }  
 **System**.out.println(user.getId());  
 **System**.out.println(user.getUsername());  
 **System**.out.println(user.getPassword());  
 **System**.out.println(user.getBrityday());  
 user.setId("1");  
 **return** user;  
}

控制台正常运行，控制台中给出提示may not be empty。

#### 四、用户信息修改和删除服务

##### 1.常用的验证注解



编写修改测试方法：

@Test  
**public void** whenUpdateSuccess() **throws Exception**{  
 **Date** date = **new** Date(**LocalDateTime**.now().plusYears(1).atZone(ZoneId.systemDefault()).toInstant().toEpochMilli()); //表示一年后的时间  
 **System**.out.println(date.getTime());  
 **String** content="{\"id\":\"1\",\"username\":\"tom\",\"password\":null,\"brityday\":"+date.getTime()+"}";  
 **String** result= mockMvc.perform(MockMvcRequestBuilders.put("/user/1")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8)  
 .content(content))  
 .andExpect(MockMvcResultMatchers.status().isOk())  
 .andExpect((MockMvcResultMatchers.jsonPath("$.id").value("1")))  
 .andReturn().getResponse().getContentAsString();  
 **System**.out.println(result);  
}

在User实体来的brithday字段上加入@Past表示判断该字段为历史时间字段。

在UserController中添加update实现方法：

@PutMapping("/{id:\\d+}")  
**public User** update(@Valid @RequestBody **User** user, BindingResult errors) {  
 **if** (errors.hasErrors()) {  
 errors.getAllErrors().stream().forEach(error -> {  
 **System**.out.println(error.getDefaultMessage());  
 });  
 }  
 **System**.out.println(user.getId());  
 **System**.out.println(user.getUsername());  
 **System**.out.println(user.getPassword());  
 **System**.out.println(user.getBrityday());  
 user.setId("1");  
 **return** user;  
}

运行测试方法，控制台中输出：

1603086161884

may not be empty

must be in the past

1

tom

null

Mon Oct 19 13:42:41 CST 2020

{"id":"1","username":"tom","password":null,"brityday":1603086161884}

##### 2.自定义消息

上述控制台提供的错误提示是may not be empty must be in the past，那么我们现在自定义这个消息。

在错误提示代码中修改方法：

**if** (errors.hasErrors()) {  
 errors.getAllErrors().stream().forEach(error -> {  
 **FieldError** fieldError = (**FieldError**)error;  
 **String** message = fieldError.getField() +" "+  
 error.getDefaultMessage();  
 **System**.out.println(message);  
 //System.out.println(error.getDefaultMessage());  
 });  
}

这样就把将某个字段报错的名字给打印出来：

password may not be empty

brityday must be in the past

那么我们在对这两个字段属性进行修改，进入User实体类，找到这两个字段的注解，因为这些注解都是继承了同一个包里，且他们都有共同的属性message，所以我们自定义错误提示信息如下：

@NotBlank (message="密码不能为空！") //不为空  
**private String** password;  
  
@Past(message="生日必须是历史时间！") //过去的时间  
**private Date** brityday;

在次运行测试方法，控制台就会输出：

密码不能为空！

生日必须是历史时间！

##### 3.自定义校验注解

在com.zhkf包下新建validator包，包下建立MyConstraint接口

@Target({ElementType.METHOD,ElementType.FIELD}) //自定义标注在方法与字段上面  
@Retention(RetentionPolicy.RUNTIME) //运行时的注解  
@Constraint(validatedBy = **MyConstraintValidator**.**class**) //校验逻辑注解  
**public** @**interface** MyConstraint {  
 **String** message() **default** "{org.hibernate.validator.constraints.NotBlank.message}";  
  
 **Class**<?>[] groups() **default** {};  
  
 **Class**<? **extends** Payload>[] payload() **default** {};  
  
}

同时新建接口的逻辑校验类MyConstraintValidator实现ConstraintValidator的泛型接口

**public class MyConstraintValidator implement** ConstraintValidator<MyConstraint,**Object**> {  
 @Autowired  
 **private** HelloService helloService;  
 @Override  
 **public void** initialize(MyConstraint myConstraint) { //校验器初始化  
 **System**.out.println("校验初始化...");  
 }  
  
 @Override  
 **public boolean** isValid(**Object** value, ConstraintValidatorContext constraintValidatorContext) { //校验逻辑  
 helloService.greeting("tom");  
 **System**.out.println(value);  
 **return false**;  
 }

在建立一个测试接口，用于参数的校验，com.zhkf包下建立service包创建HelloService接口：

**public interface** HelloService {  
  
 **String** greeting(**String** name);  
}

同目录下新建impl的实现类HelloServiceImpl:

@Service  
**public class HelloServiceImpl implements** HelloService {  
 @Override  
 **public String** greeting(**String** name) {  
 **System**.out.println("greeting");  
 **return** "hello "+name;  
 }

}

在User实体类的username属性加入我们自定义的注解：

@MyConstraint(message = "这是一个测试注解")  
**private String** username;

运行更新的测试方法，查看打印输出结果为：

1603088701298

校验初始化...

greeting

tom

这是一个测试注解

生日必须是历史时间！

密码不能为空！

1

tom

null

Mon Oct 19 14:25:01 CST 2020

{"id":"1","username":"tom","password":null,"brityday":1603088701298}

##### 4.删除服务

测试为先，先写测试方法：

//删除  
@Test  
**public void** whenDeleteSuccess() **throws Exception**{  
 mockMvc.perform(MockMvcRequestBuilders.delete("/user/1")  
 .contentType(**MediaType**.APPLICATION\_JSON\_UTF8))  
 .andExpect(MockMvcResultMatchers.status().isOk());  
  
}

删除服务方法：

//删除服务  
@DeleteMapping("/{id:\\d+}")  
**public void** delete(@PathVariable **String** id){  
 **System**.out.println(id);  
}

#### 五、RESTfulAPI错误处理

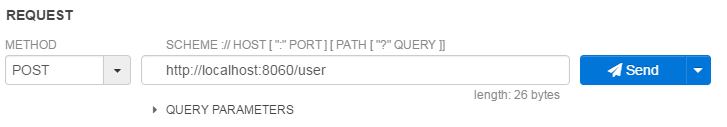
###### 1.Spring Boot中默认的错误处理机制

安装Restlet Client插件(安装步骤参见百度)，

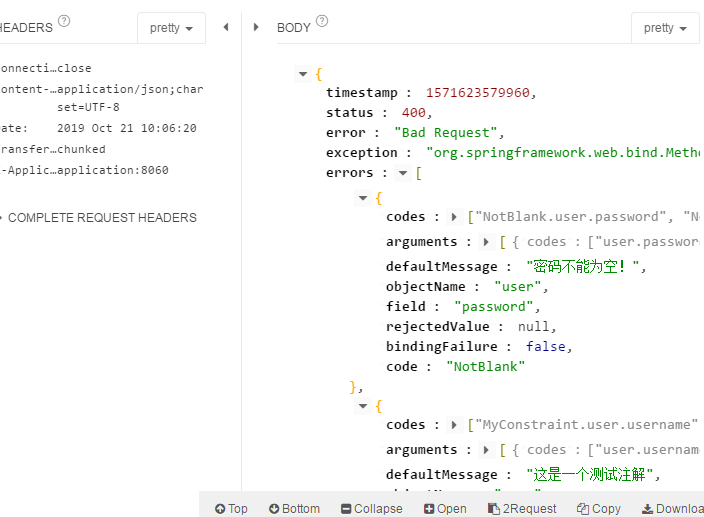
BasicErrotController控制器的作用，当是浏览器发送的请求时，将会返回html,不是浏览器发送的请求返回json字符。修改create()服务，将异常处理注释掉：

@PostMapping  
**public User** create(@Valid @RequestBody **User** user*/\*, BindingResult errors\*/*) {  
  
 */\* if (errors.hasErrors()) {  
 errors.getAllErrors().stream().forEach(error -> System.out.println(error.getDefaultMessage()));  
 }\*/*

Restlet Client:



结果：

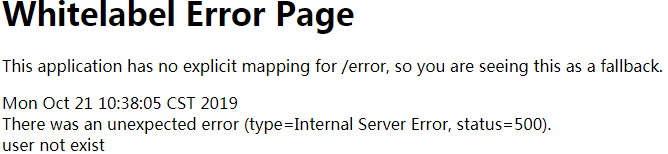


在测试一个异常：

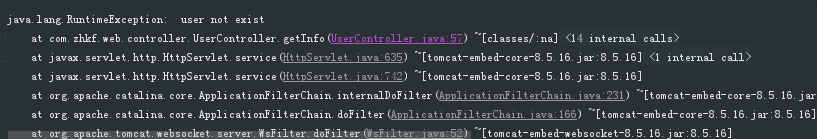
将getInfo()方法修改如下：

@GetMapping("/{id:\\d+}")  
@JsonView(**User**.UserDetailView.**class**)  
**public User** getInfo(@PathVariable **String** id) {  
 **throw new** RuntimeException(" user not exist");  
 */\*User user = new User();  
 user.setUsername("tom");  
 return user;\*/*}

从新启动项目，浏览器输入<http://localhost:8060/user/1>或测试Restlet Client调试插件：



控制台报错：

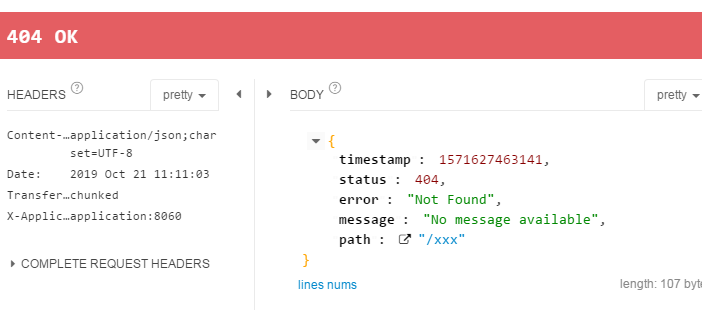


SpringBoot的错误处理机制是依据状态码来处理的，我们也可以通过自定义来完成信息的处理。

###### 2自定义异常处理

(1).在resources创建/resources/error文件夹，文件夹下创建404.html文件，自定义文件内容，从新运行项目，输入一个该服务下不存在的路径，页面将根据状态码，返回想要的错误提示页面，如：405内部服务异常。但是Restlet Client返回的仍是Json错误提示格式，因为SpringBoot将错误信息分为两类，一类是html错误格式，返回给浏览器，Json的错误格式返回给如Restlet Clien的插件格式。所以自定义的html页面只对浏览器起作用。如下是我自定义的404错误界面，两者的错误提示对比：

GMS6CR`2UHY@FUD5C8FK6GM.png



(2).com.zhkf下新建exception.UserNotExistException.java类并继承RuntimeException

**public class UserNotExistException extends RuntimeException** {  
 **private String** id;  
 **public** UserNotExistException(**String** id){  
 **super**("user not exist");  
 **this**.id=id;  
 }  
 **public String** getId() {  
 **return** id;  
 }  
 **public void** setId(**String** id) {  
 **this**.id = id;  
 }  
}

UserController类的getInfo():

@GetMapping("/{id:\\d+}")  
@JsonView(**User**.UserDetailView.**class**)  
**public User** getInfo(@PathVariable **String** id) {  
 **throw new** UserNotExistException(id);}

在编写一个异常处理类：controller文件夹下创建ControllerExceptionHandler.java

@ControllerAdvice //表示这个类里的方法都是处理其它controller所抛出的异常  
**public class ControllerExceptionHandler** {  
  
 @ExceptionHandler(**UserNotExistException**.**class**)  
 @ResponseBody //返回Json格式  
 @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR) //返回500的一个错误码  
 **public** Map<**String**,**Object**> handleUserNotExistException(**UserNotExistException** ex){  
 Map<**String**,**Object**> result = **new** HashMap<>();  
 result.put("id",ex.getId());  
 result.put("message",ex.getMessage());  
 **return** result;  
 }  
}

从新启动程序，Restlet Client插件调试<http://localhost:8060/user/1>,查看结果为：



#### 六、RESTful API的拦截

需求：记录所有服务的错误时间

##### 1.过滤器(Filter)

**第一层:过滤器Demo**

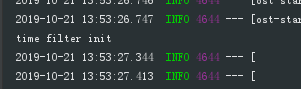
Web文件下新建filter文件夹，该文件夹下建TimeFilter.java类，实现Filter方法添加@Component注解：

@Component  
**public class TimeFilter implements** Filter {  
 @Override  
 **public void** init(FilterConfig filterConfig) **throws ServletException** {  
 **System**.out.println("time filter init");  
 }  
  
 @Override  
 **public void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws IOException**, **ServletException** {  
 **System**.out.println("time filter start");  
 **long** start = **new** Date().getTime();  
 filterChain.doFilter(servletRequest,servletResponse);  
 **System**.out.println("time filter 耗时:"+(**new** Date().getTime()-start));  
 **System**.out.println("time filter finish");  
 }  
  
 @Override  
 **public void** destroy() {  
 **System**.out.println("time filter destroy");  
 }

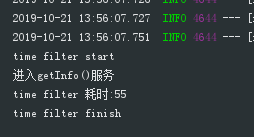
UserCntroller中的getInfo()方法改成如下：

@GetMapping("/{id:\\d+}")  
@JsonView(**User**.UserDetailView.**class**)  
**public User** getInfo(@PathVariable **String** id) {  
 **System**.out.println("进入getInfo()服务");  
 **User** user = **new** User();  
 user.setUsername("tom");  
 **return** user;  
}

启动项目程序：



浏览器或调试插件输入<http://localhost:8060/user/1>。控制台输出内容如下：



第二层：将自定义的过滤器的@Component注解去掉，使其成为一个第三方过滤器的形式加入到项目中。web目录下新建config.WebConfig.java,配置代码如下：

@Configuration  
**public class WebConfig** {  
  
 @Bean  
 **public FilterRegistrationBean** timeFilter(){  
 **FilterRegistrationBean** registrationBean=**new** FilterRegistrationBean();  
 **TimeFilter** timeFilter = **new** TimeFilter();  
 registrationBean.setFilter(timeFilter);  
 List<**String**> urls = **new** ArrayList<>();  
 urls.add("/\*");  
 registrationBean.setUrlPatterns(urls);  
 **return** registrationBean;  
 }  
}

这种实现出和第一层一样的过滤效果了。

##### 2.拦截器(Interceptor)

com.zhkf文件路径下新建interceptor.TimeInteceptor.java类实现HandlerInterceptor:

@Component  
**public class TimeInteceptor implements** HandlerInterceptor {  
  
 //在控制器方法之前调用  
 @Override  
 **public boolean** preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, **Object** handle) **throws Exception** {  
 **System**.out.println("preHandle");  
 httpServletRequest.setAttribute("strartTime",**new** Date().getTime());  
  
 **System**.out.println(((**HandlerMethod**)handle).getBean().getClass().getName());  
 **System**.out.println(((**HandlerMethod**)handle).getMethod().getName());  
 **return true**;  
 }  
  
 //在控制器方法之后调用，如果控制器里抛出异常，这个方法就不会被调用了  
 @Override  
 **public void** postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, **Object** o, **ModelAndView** modelAndView) **throws Exception** {  
 **System**.out.println("postHandle");  
 **Long** start = (**long**)httpServletRequest.getAttribute("startTime");  
 **System**.out.println("time interceptor 耗时:"+(**new** Date().getTime()-start));  
 }  
  
 //在控制器方法之后调用，无论控制器是否抛出异常，这个方法都会被调用  
 @Override  
 **public void** afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, **Object** o, **Exception** e) **throws Exception** {  
 **System**.out.println("pafterCompletion");  
 **Long** start = (**long**)httpServletRequest.getAttribute("startTime");  
 **System**.out.println("time interceptor 耗时:"+(**new** Date().getTime()-start));  
 **System**.out.println("e is "+e);  
 }  
}

WebConfig.java继承WebMvcConfigurerAdapter,添加代码：

@Autowired  
**private TimeInteceptor** timeInteceptor;  
//添加拦截配置  
@Override  
**public void** addInterceptors(**InterceptorRegistry** registry) {  
 registry.addInterceptor(timeInteceptor);  
}

完成拦截器的Demo配置。

##### 4.切片(Aspect)

pom.xml中添加SpringAOP依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-aop</**artifactId**>  
</**dependency**>

新建com.zhkf.web.aspect文件夹，创建TimeAspect.java类:

@Aspect  
@Component  
**public class TimeAspect** {  
  
 @Around("execution(\* com.zhkf.web.controller.UserController.\*(..))")  
 **public Object** handleControllerMethod(ProceedingJoinPoint proceedingJoinPoint) **throws Throwable**{ //表示当前拦截的方法的一个对象  
 **System**.out.println("tome aspect start");  
 **long** start =**new** Date().getTime();  
 **Object** object= proceedingJoinPoint.proceed(); //调用控制器中被拦截的方法  
 **System**.out.println("time Aspect 耗时:"+(**new** Date().getTime()-start));  
 **System**.out.println("time aspect end");  
 **return Object**;  
 }  
}

总结：

过滤器(Filter)：可以拿到原始的http请求和响应的信息，但是拿不到处理请求的方法。

拦截器(Interceptor):既可以拿到原始的http请求和响应，也能拿到处理请求的方法，但是拿不到方法里的参数。

切片(Aspect):可以拿到方法参数里的值，但是拿不到原始的http的请求何响应的对象。

顺序：

Filter-->Interceptor-->Aspect-->Controller

当服务方法抛出异常时：Controller-->Aspect->ControllerAdvice-->Intercceptor-->Filter

#### 七、文件上传与下载

##### 1.文件上传

测试类：

//模拟文件上传  
@Test  
**public void** whenUploadSuccess() **throws Exception**{  
  
 **String** result= mockMvc.perform(MockMvcRequestBuilders.fileUpload("/file")  
 .file(**new** MockMultipartFile("file","test.exe","multipart/form-data","hello upload".getBytes("UTF-8"))))  
 .andExpect(MockMvcResultMatchers.status().isOk())  
 .andReturn().getResponse().getContentAsString();  
  
 **System**.out.println(result);  
}

Controller目录下新建FileController.java

@RestController  
@RequestMapping("/file")  
**public class FileController** {  
  
 **private String** folder = "G:\\yjxm\\zhkf-security-demo\\src\\main\\java\\com\\zhkf\\web\\controller";  
  
 @PostMapping  
 **public FileInfo** upload(MultipartFile file) **throws Exception** {  
 **System**.out.println(file.getName());  
 **System**.out.println(file.getOriginalFilename());  
 **System**.out.println(file.getSize());  
  
 **File** localFile = **new** File(folder, **new** Date().getTime() + ".txt");  
 //file.getInputStream(); //将文件发送到第三方存储里面  
 file.transferTo(localFile);  
 **return new** FileInfo(localFile.getAbsolutePath());  
 }

}

dto下新建FileInfo:

**public class FileInfo** {  
  
 **private String** path;  
 **public** FileInfo(**String** path){  
 **this**.path=path;  
 }  
  
 **public String** getPath() {  
 **return** path;  
 }  
  
 **public void** setPath(**String** path) {  
 **this**.path = path;  
 }  
}

##### 2.文件下载

Pom.xml文件加入commons-io依赖：

<**dependency**>  
 <**groupId**>commons-io</**groupId**>  
 <**artifactId**>commons-io</**artifactId**>  
</**dependency**>

FileController添加下载实现：

@GetMapping("/{id}")  
**public void** download(@PathVariable **String** id, HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) **throws Exception**{  
  
 **try**(  
 InputStream inputStream = **new** FileInputStream(**new** File(folder,id+".txt"));  
 OutputStream outputStream = httpServletResponse.getOutputStream();  
 ){  
 httpServletResponse.setContentType("application/x-download");  
 httpServletResponse.addHeader("Content-Disposition","attachment;filename=test.txt");  
 **IOUtils**.copy(inputStream,outputStream);  
 outputStream.flush();  
 }  
}

#### 八、异步处理服务

###### 1.使用Runnable异步处理Rest服务、

简单同步处理：web下新建async.AsyncController.java

@RestController  
**public class AsyncController** {  
  
 **private** Logger logger= **LoggerFactory**.getLogger(getClass());  
  
 @RequestMapping("/order")  
 **public String** order() **throws Exception**{  
 logger.info("主线程开始...");  
 **Thread**.sleep(1000);  
 logger.info("主线程结束...");  
 **return** "success";  
 }

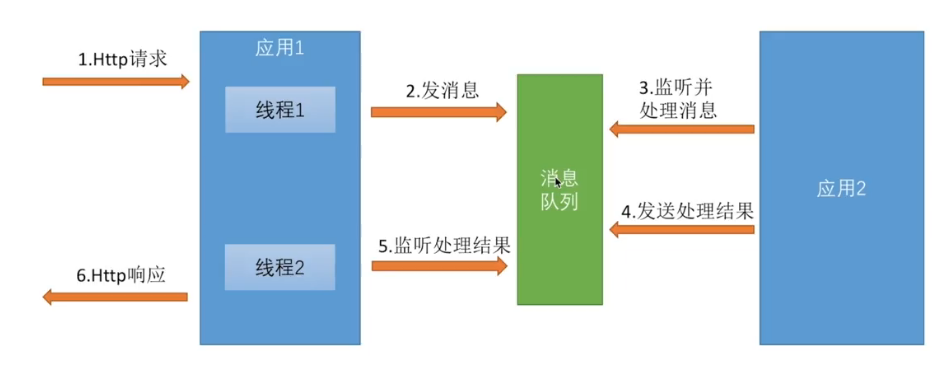
异步改编：

@RestController  
**public class AsyncController** {  
  
 **private** Logger logger= **LoggerFactory**.getLogger(getClass());  
  
 @RequestMapping("/order")  
 **public** Callable<**String**> order() **throws Exception**{  
 logger.info("主线程开始...");  
 Callable<**String**> result=**new** Callable<**String**>(){  
 @Override  
 **public String** call() **throws Exception** {  
 logger.info("副线程开始...");  
 **Thread**.sleep(1000);  
 logger.info("副线程结束...");  
 **return** "success";  
 }  
 };  
 **Thread**.sleep(1000);  
 logger.info("主线程结束...");  
 **return** result;  
 }  
}

用Runnable的异步处理，父线程是由主线程调起得。

###### 2.使用DeferredResult异步处理Rest服务

例子：



###### Async文件下新建DeferredResultHolder.java，这个类主要是将线程1与线程2之间的信息进行交互

@Component  
**public class DeferredResultHolder** {  
  
 **private** Map<**String**, **DeferredResult**<**String**>> map = **new** HashMap<**String**, **DeferredResult**<**String**>>();  
 **public** Map<**String**, **DeferredResult**<**String**>> getMap() {  
 **return** map;  
 }  
 **public void** setMap(Map<**String**, **DeferredResult**<**String**>> map) {  
 **this**.map = map;  
 }  
}

新建MockQueue.java用来模拟队列对象

@Component  
**public class MockQueue** {  
 **private String** placeOrder; //模拟订单下单的消息  
 **private String** completeOrder; //模拟订单完成的消息  
  
 **public String** getPlaceOrder() {  
 **return** placeOrder;  
 }  
  
 **public void** setPlaceOrder(**String** placeOrder) **throws Exception**{  
 **System**.out.println("接到下单请求， "+placeOrder);  
 **Thread**.sleep(1000);  
 **this**.completeOrder=placeOrder;  
 **System**.out.println("下单请求处理完毕，"+placeOrder);  
 }  
  
 **public String** getCompleteOrder() {  
 **return** completeOrder;  
 }  
  
 **public void** setCompleteOrder(**String** completeOrder) {  
 **this**.completeOrder = completeOrder;  
 }  
}

消息队列接听器QueueListener：

@Component  
**public class QueueListener implements** ApplicationListener<**ContextRefreshedEvent**> {  
  
 @Autowired  
 **private MockQueue** mockQueue;  
  
 @Autowired  
 **private DeferredResultHolder** deferredResultHolder;  
  
 **private** Logger logger = **LoggerFactory**.getLogger(getClass());  
  
 @Override  
 **public void** onApplicationEvent(**ContextRefreshedEvent** contextRefreshedEvent)  
 {  
 **new** Thread(()->{  
 **while** (**true**){  
 **if**(**StringUtils**.isNotBlank(mockQueue.getCompleteOrder())){  
 **String** orderNumber = mockQueue.getCompleteOrder();  
 logger.info("返回订单处理结果："+orderNumber);  
 deferredResultHolder.getMap().get(orderNumber).setResult("place order success");  
 mockQueue.setCompleteOrder(**null**);  
 }**else**{  
 **try** {  
 **Thread**.sleep(100);  
 }**catch** (**Exception** e){  
  
 }  
 }  
  
 }  
 }).start();  
 }  
}

在线程服务类中AsyncController中修改

@RequestMapping("/order")  
 **public DeferredResult**<**String**> order() **throws Exception**{  
 logger.info("主线程开始...");  
 **String** orderNumber = **RandomStringUtils**.randomNumeric(8);  
 mockQueue.setPlaceOrder(orderNumber);  
 **DeferredResult**<**String**> result=**new** DeferredResult<>();  
 deferredResultHolder.getMap().put(orderNumber,result);  
 logger.info("主线程结束...");  
 **return** result;  
 }

###### 3.异步处理的配置

WebMvcConfigurerAdapter中有一个

@Override  
**public void** configureAsyncSupport(**AsyncSupportConfigurer** configurer) {  
 configurer.registerCallableInterceptors();

//configurer.registerDeferredResultInterceptors();

}

这个方法是拦截异步线程时使用的。

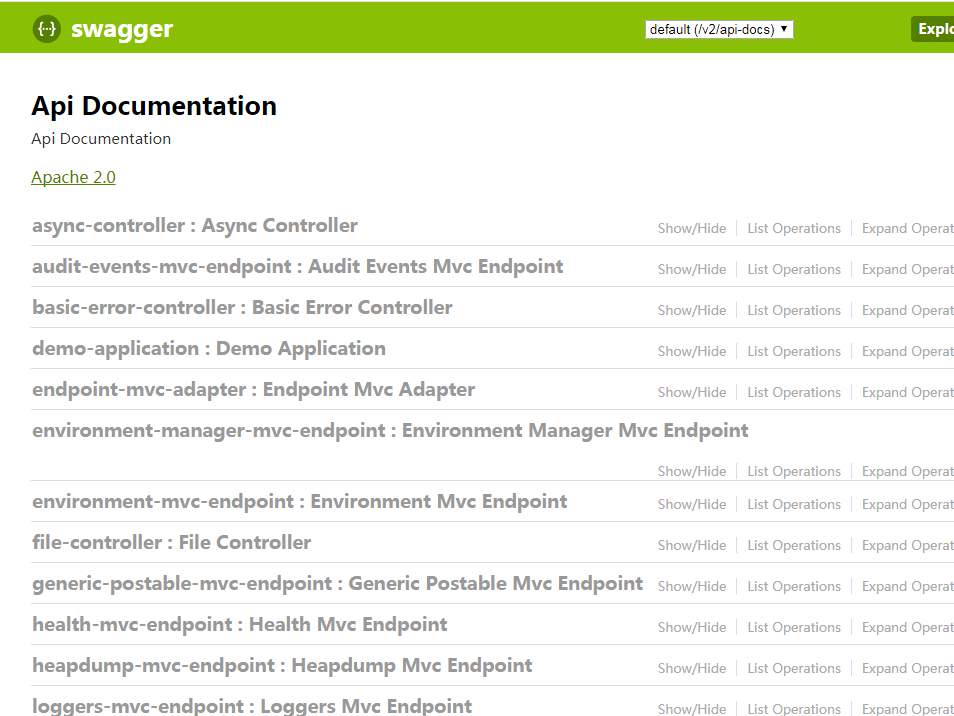
#### 九、与前端开发并行处理

##### 1.使用swagger自动生成html文档

pom.xml中添加依赖：

<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger2</**artifactId**>  
 <**version**>2.7.0</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>io.springfox</**groupId**>  
 <**artifactId**>springfox-swagger-ui</**artifactId**>  
 <**version**>2.7.0</**version**>  
</**dependency**>

项目入口添加注解@EnableSwagger2,启动项目访问<http://localhost:8060/swagger-ui.html>



在controller方法上添加注解

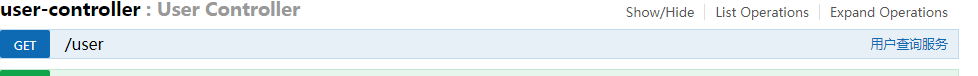
@ApiOperation(value="用户查询服务") //swagger-ui中返回方法名

参数名自定义，如果参数是实体类，可在实体类的属性上添加注解

@ApiModelProperty(value="用户名称")  
**private String** username;  
@ApiModelProperty(value="用户年龄起始值") //Swagger-ui中返回参数名  
**private int** age;  
@ApiModelProperty(value="用户年龄终止值")  
**private int** ageTo;

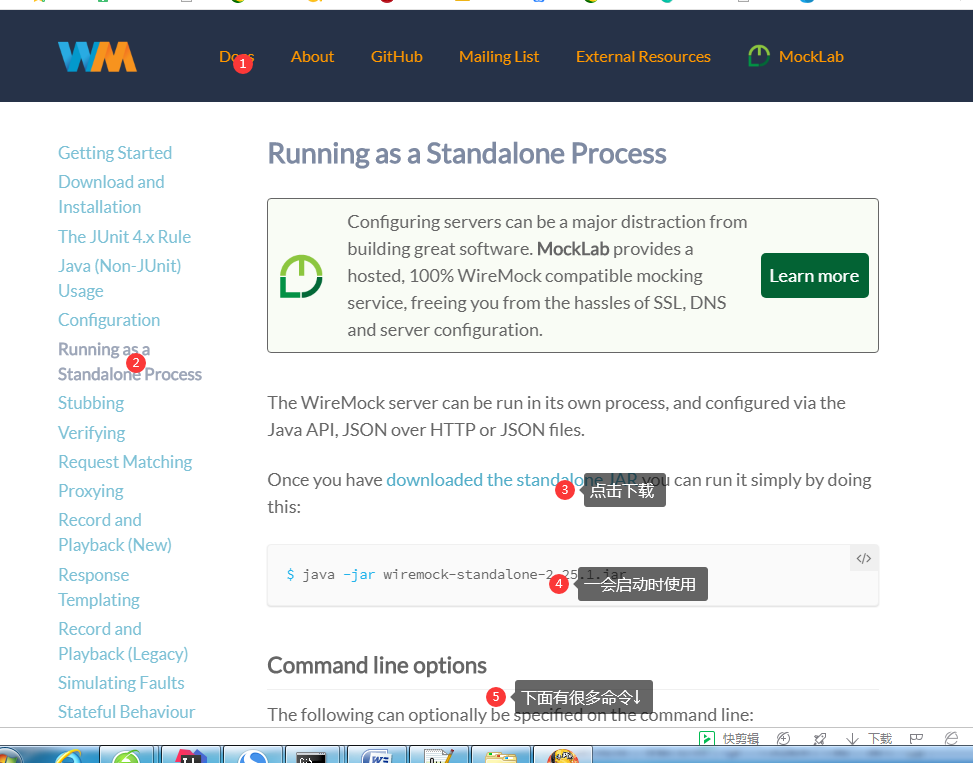
如果参数是单独属性，直接在参数类型前面添加@ApiParam(“参数介绍”)

效果图：



##### 2.使用WireMock快速伪造ReSTful服务

百度搜wrieMock,进入官网



### 第四章 SpringSecurity开发基于表单的认证

##### 一、简介

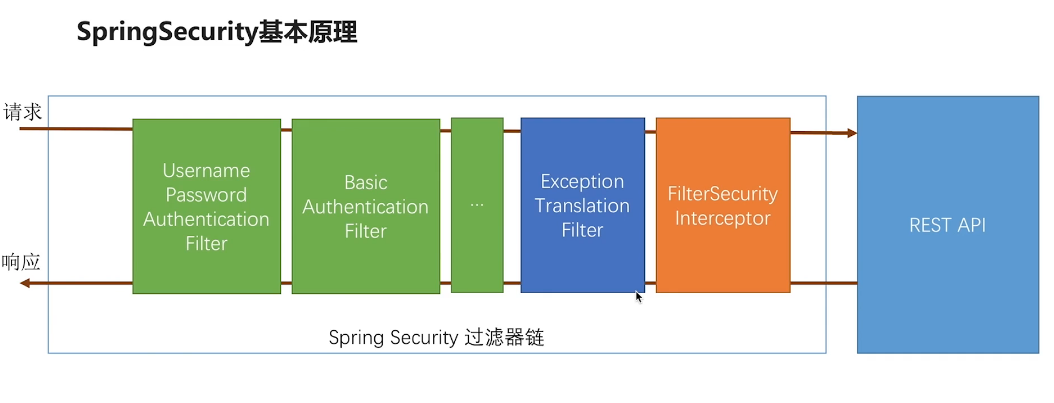
SpringSecurity核心功能：

认证（你是谁）

授权（你能干什么）

攻击防护（防止伪造身份）

##### 二、SpringSecurity基本原理



##### 三、实现用户名+密码认证

###### 1.处理用户信息获取逻辑

Com.zhkf.security.browser下新建MyUserDetailsService.java实现UserDetailsService接口

@Override  
**public** UserDetails loadUserByUsername(**String** username) **throws UsernameNotFoundException** {  
  
 logger.info("登录用户名："+username);  
  
 **String** password = 123456;  
 logger.info("密码是："+password);  
 //根据用户名查找用户信息  
 **return new** User(username,password,AuthorityUtils.commaSeparatedStringToAuthorityList("admin")); //写死的密码与权限  
}

###### 2.处理用户校验逻辑

@Override  
**public** UserDetails loadUserByUsername(**String** username) **throws UsernameNotFoundException** {  
 logger.info("登录用户名："+username);  
 **String** password = "123456";  
 logger.info("密码是："+password);  
 //根据用户名查找用户信息  
 **return new** User(username,password,**true**,**true**,**true**,**true**, AuthorityUtils.commaSeparatedStringToAuthorityList("admin")); //写死的密码与权限  
}

###### 3.处理密码加密解密

BrowserSercurityConfig.java添加PasswordEncoder方法：

@Bean  
**public** PasswordEncoder passwordEncoder(){  
 **return new** BCryptPasswordEncoder();  
}

MyUserDetailsService代码更改：

**private** Logger logger = **LoggerFactory**.getLogger(getClass());  
@Autowired  
**private** PasswordEncoder passwordEncoder;  
@Override  
**public** UserDetails loadUserByUsername(**String** username) **throws UsernameNotFoundException** {  
 logger.info("登录用户名："+username);  
 **String** password = passwordEncoder.encode("123456");  
 logger.info("密码是："+password);  
 //根据用户名查找用户信息  
 **return new** User(username,password,**true**,**true**,**true**,**true**, AuthorityUtils.commaSeparatedStringToAuthorityList("admin")); //写死的密码与权限  
}

##### 四、个性化用户认证

###### 1.自定义登录界面

http.formLogin()后面添加登录首页面.loginPage(“/zhkf-login.html”)，并添加表单跳转路径：

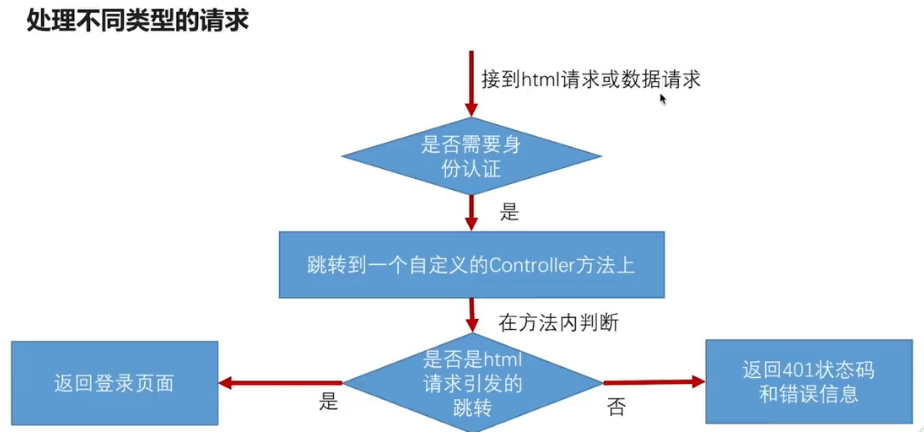
.antMatchers("/zhkf-login.html").permitAll()

.loginProcessingUrl("/authentication/form")

resources下新建resources.zhkf-login.html<**h1**>登录界面</**h1**>  
<**form** action="/authentication/form" method="post">  
 <**table**>  
 <**tr**>  
 <**td**>用户名：</**td**>  
 <**td**><**input** type="text" name="username"></**td**>  
 </**tr**>  
 <**tr**>  
 <**td**>密码：</**td**>  
 <**td**><**input** type="password" name="password"></**td**>  
 </**tr**>  
 <**tr**>  
 <**td** colspan="2"><**button** type="submit">登录</**button**></**td**>  
 </**tr**>  
 </**table**>  
</**form**>

这个时候点击登录跳转后报403错误，这是被SpringSecurity中自带的跨域请求拦截了，所以需要把csrf()禁掉，只需在代码中添加：

.and()  
 .csrf().disable(); //将自带的跨域请求禁调



在com.zhkf.security.browser中新建BrowserSecurityController.java控制判断

(1).写一个判断的方法

@RequestMapping("/authentication/require")  
**public String** requireAuthentication(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) **throws Exception**{  
 **return null;**  
}

(2).将BrowserSecurityConfig的.loginPage(“/zhkf-login.html”)改为和控制层传来的路径相同.loginPage(“/authentication/require”)

(3).在回到BrowserSecurityController中做判断，添加：

**private** RequestCache requestCache = **new** HttpSessionRequestCache();

拿到当前的请求缓存到session中去，在执行身份认证方法时，在把session中的数据拿过来

(4).修改requireAuthentication 方法，将请求和响应传进去：

SavedRequest savedRequest = requestCache.getRequest(httpServletRequest,httpServletResponse);

(5).做跳转，添加

**private** RedirectStrategy redirectStrategy = **new** DefaultRedirectStrategy();

(6).判断操作，在requireAuthentication方法中添加：

**if**(savedRequest != **null**){  
 **String** targetUrl = savedRequest.getRedirectUrl();  
 logger.info("引发跳转的请求是："+targetUrl);  
 **if**(StringUtils.endsWithIgnoreCase(targetUrl,".html")){  
 redirectStrategy.sendRedirect(httpServletRequest,httpServletResponse,securityProperties.getBroweserProperties().getLoginPage());  
 }  
}

(7).添加状态码方法上添加：

@ResponseStatus(code = HttpStatus.UNAUTHORIZED)

(8).定义返回类型：

新建com.zhkf.security.browser.support.SimpleResponse.java

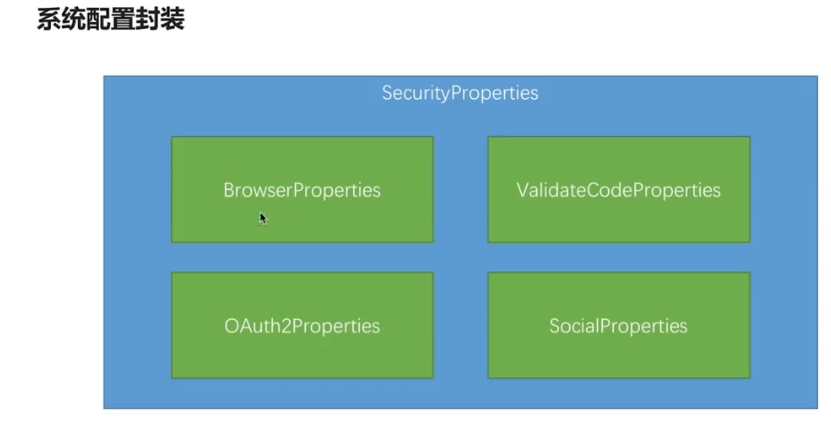
**public** SimpleResponse(**Object** content){  
 **this**.content=content;  
}  
**private Object** content;  
**public Object** getContent() {  
 **return** content;  
}  
**public void** setContent(**Object** content) {  
 **this**.content = content;  
}

(9).zhkf-security-demo的配置文件中添加：

**zhkf.security.browser.loginPage**=/demo-login.html

(10).在resources下创建demo-login.html

(11).系统配置封装



在core模块中新建com.zhkf.security.core.properties.BrowserProperties.java与SecurityProperties.java类。

SecurityProperties.java中声明BrowserProperties对象，并添加@ConfigurationProperties注解

@ConfigurationProperties(prefix = "zhkf.security")  
**public class SecurityProperties** {  
 **private BrowserProperties** browser = **new** BrowserProperties();  
  
 **public BrowserProperties** getBrowserProperties() {  
 **return** browser;  
 }  
  
 **public void** setBrowserProperties(**BrowserProperties** browser) {  
 **this**.broweser = browser;  
 }  
}

在BrowserProperties中添加loginPage属性，并生成get\set方法。

新建com.zhkf.security.core.SecurityCoreConfig.java:

@Configuration  
@EnableConfigurationProperties(**SecurityProperties**.**class**)  
**public class SecurityCoreConfig** {  
  
}

(12).在将SecurityProperties注入到BrowserSecurityController.java中去

@Autowired  
**private SecurityProperties** securityProperties;

(13).将BrowserProperties中的loginPage属性设置确定值：

Private String loginPage=”/zhkf-login.html”;

(14).在BrowserSecurityConfig中将SecurityProprties加入到授权验证中去：

@Autowired  
**private SecurityProperties** securityProperties;

--------------------------------------------------------------------------------------------

.antMatchers("/authentication/require",securityProperties.getBroweserProperties().getLoginPage()).permitAll()

这里最后还需要在core模块中加入

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-configuration-processor</**artifactId**>  
</**dependency**>

这里一定要注意声明的browser一定要与demo模块中的配置名相同，不然会找不到路径。

###### 2.自定义登录成功处理

(1).Borwser模块security.browser下创建authentication.ZhkfAuthenticationSuccessHandler.java

实现AuthenticationSuccessHandle：

**private** Logger logger= **LoggerFactory**.getLogger(getClass());  
  
@Autowired  
**private ObjectMapper** objectMapper;  
  
//登录成功时调用  
@Override  
**public void** onAuthenticationSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) **throws IOException**, **ServletException** {  
  
 logger.info("登录成功");  
 httpServletResponse.setContentType("application/json;charset=UTF-8");  
 httpServletResponse.getWriter().write(objectMapper.writeValueAsString(authentication));  
}

并在类名声明@Compenent(“zhkfAuthenticationSuccessHandler”)

(2).在BrowserSecurityConfig下注入：

@Autowired  
**private** AuthenticationSuccessHandler zhkfAuthenticationSuccessHandler;

注意这个zhkfAuthenticationSuccessHandler要与上面@Compenent的参数一致

(3).在登录表单url的配置后面添加

.successHandler(zhkfAuthenticationSuccessHandler)

表示登录成功后要进入这个方法模拟认证。

###### 3.自定义登录失败处理

以相同的步骤实现登陆失败处理，只是创建的类要实现AuthenticationFilureHandler类，部分代码：

logger.info("登录失败");  
httpServletResponse.setStatus(HttpStatus.INTERNAL\_SERVER\_ERROR.value());  
httpServletResponse.setContentType("application/json;charset=UTF-8");  
httpServletResponse.getWriter().write(objectMapper.writeValueAsString(e));

###### 4.代码重构

(1).core模块的properties文件下新建枚举类LoginType

REDIRECT, //跳转  
JSON //返回JSON

BrowserProperties中添加：

**private** LoginType loginType = LoginType.JSON;

并生成Get和Set方法

(2).ZhkfAuthenticationSuccHandler.java将实现AuthenticationSuccessHandler注释掉，继承

SavedRequestAwarAuthenticationSuccesssHandler,并注入SecurityProperties：

@Autowired  
**private SecurityProperties** securityProperties;  
  
//登录成功时调用  
@Override  
**public void** onAuthenticationSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) **throws IOException**, **ServletException** {  
  
 logger.info("登录成功");  
  
 **if**(LoginType.JSON.equals(securityProperties.getBrowser().getLoginType())){  
 httpServletResponse.setContentType("application/json;charset=UTF-8");  
 httpServletResponse.getWriter().write(objectMapper.writeValueAsString(authentication));  
 }**else**{  
 **super**.onAuthenticationSuccess(httpServletRequest,httpServletResponse,authentication);  
 }  
}

同理修改登录失败方法，但失败类中继承的是SimpleUrlAUthenticationFailureHandler。

由于**private** LoginType loginType = LoginType.JSON;

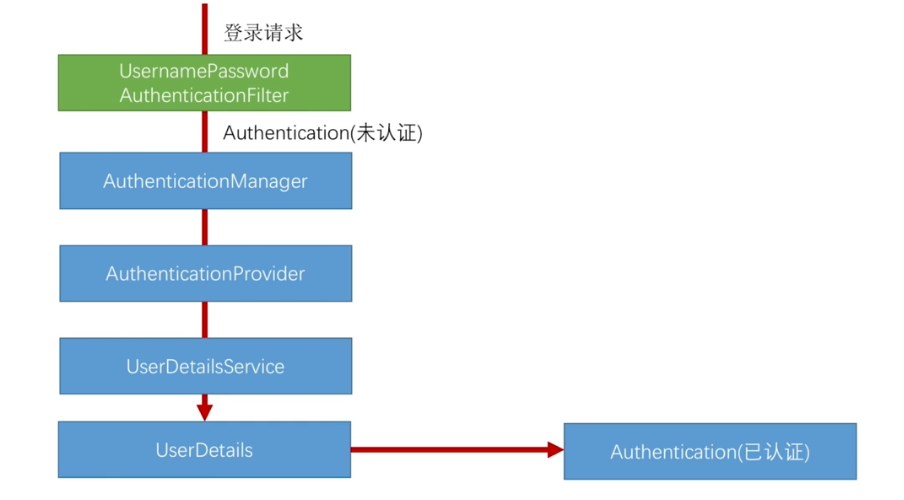
无论登录成功还是失败都会返回JSON,这里我们就可以自定义返回结果：

在demo模块的配置文件中添加：

**zhkf.security.browser.loginType**=REDIRECT

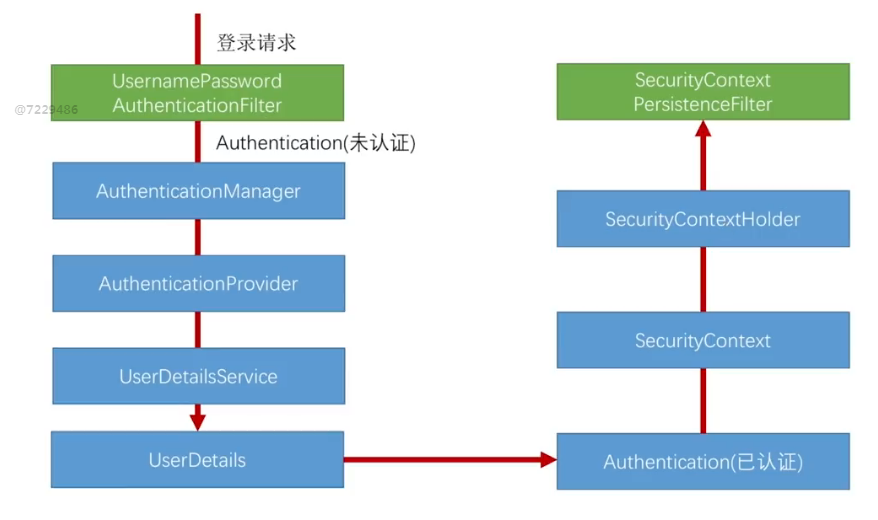
并在resourcce中创建index.html。这样我们访问index.html请求时就会跳转到该html下，如果没有该页面，就会返回401错误。

##### 五、认证流程

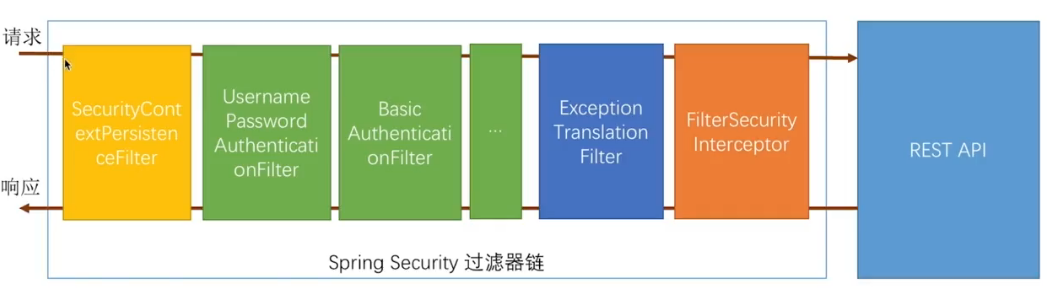


进行登录请求后，首先经过的是UserPasswordFilter类，他获取到请求中携带的用户名密码，他拿用户名密码构造了一个叫UsernamePasswordAuthenticationToken(Authentication的一个实现)的对象。经过setDetails将请求的信息映射(包含ip、session等)到UsernamePasswordAuthenticationToken对象中去。

下面就经过AuthenticationManager,这个类本身不做任何处理，它把校验的的逻辑全都交给了AuthenticationProvider处理。它调用UserDetailsService获得用户信息，最后把用户信息拼装到Authentication。Authentication在将认证的信息返回回去。



在认证器执行成功之前，SecurityContext会把认证信息应用到上下文中，然后可以通过SecurityContextHolder获取认证信息，最后经过SecurityContextPersistenceFilter检测认证信息，如果有就放在session中去。当进入UserPasswordAuthenticationFilter对象时，也会先经过SecurityContextPersistenceFilte进行线程的检查，如果有就放进session中去



获取认证信息

**SecurityContextHolder**.getContext().getAuthentication()

也可以直接通过Authentication返回：

Public Object getMe(Authentication authentication){

return authentication

}

##### 六、实现图形验证码功能

###### 1.开发生成图形验证码接口

Core模块新建core文件夹下新建validate.code.ImageCode.java:

**private BufferedImage** image;  
**private String** code;  
**private LocalDateTime** expireTime;  
  
  
  
**public** ImageCode(**BufferedImage** image,**String** code,**int** expireIn){  
 **this**.image=image;  
 **this**.code=code;  
 **this**.expireTime=**LocalDateTime**.now().plusSeconds(expireIn);  
}

**public** ImageCode(**BufferedImage** image, **String** code, **LocalDateTime** expireTime){  
 **this**.image = image;  
 **this**.code = code;  
 **this**.expireTime = expireTime;  
}

其他get\set方法省略

同目录下新建ValidateCodeController.java

**private static final String** SESSION\_KEY="SESSION\_KEY\_IMAGE\_CODE";  
  
 **private** SessionStrategy sessionStategy = **new** HttpSessionSessionStrategy();  
  
 @GetMapping("/code/image")  
 **public void** createCode(HttpServletRequest request, HttpServletResponse response) **throws IOException**{  
 **ImageCode** imageCode = createImageCode(request);  
 //将生成的图片放到seesion中  
 sessionStategy.setAttribute(**new** ServletWebRequest(request),SESSION\_KEY,imageCode);  
 //将生成的图片写入到响应的输出流中去  
 **ImageIO**.write(imageCode.getImage(),"JPEG",response.getOutputStream());  
 }  
  
 **private ImageCode** createImageCode(HttpServletRequest request) {  
 **int** width = 67;  
 **int** height = 23;  
 **BufferedImage** image = **new** BufferedImage(width, height, **BufferedImage**.TYPE\_INT\_RGB);  
  
 Graphics g = image.getGraphics();  
  
 **Random** random = **new** Random();  
  
 g.setColor(getRandColor(200, 250));  
 g.fillRect(0, 0, width, height);  
 g.setFont(**new** Font("Times New Roman", **Font**.ITALIC, 20));  
 g.setColor(getRandColor(160, 200));  
 **for** (**int** i = 0; i < 155; i++) {  
 **int** x = random.nextInt(width);  
 **int** y = random.nextInt(height);  
 **int** xl = random.nextInt(12);  
 **int** yl = random.nextInt(12);  
 g.drawLine(x, y, x + xl, y + yl);  
 }  
  
 **String** sRand = "";  
 **for** (**int** i = 0; i < 4; i++) {  
 **String** rand = **String**.valueOf(random.nextInt(10));  
 sRand += rand;  
 g.setColor(**new** Color(20 + random.nextInt(110), 20 + random.nextInt(110), 20 + random.nextInt(110)));  
 g.drawString(rand, 13 \* i + 6, 16);  
 }  
  
 g.dispose();  
  
 **return new** ImageCode(image, sRand, 60);  
}  
 /\*\*  
 \* 生成随机背景条纹  
 \*  
 \* @param fc  
 \* @param bc  
 \* @return  
 \*/  
 **private Color** getRandColor(**int** fc, **int** bc) {  
 **Random** random = **new** Random();  
 **if** (fc > 255) {  
 fc = 255;  
 }  
 **if** (bc > 255) {  
 bc = 255;  
 }  
 **int** r = fc + random.nextInt(bc - fc);  
 **int** g = fc + random.nextInt(bc - fc);  
 **int** b = fc + random.nextInt(bc - fc);  
 **return new** Color(r, g, b);  
 }

在BrowserSecurityConfig.java中将图片路径授权得到

.antMatchers("/authentication/require",  
 securityProperties.getBrowser().getLoginPage(),"/code/image").permitAll()

在zhkf-login.html中添加代码：

<**tr**>  
 <**td**>验证码：</**td**>  
 <**td**><**input** type="text" name="imageCode">  
 <**img** src="/code/image"></**td**>  
</**tr**>

运行即可显示验证码效果

###### 2.在认证流程中加入图形验证码校验

先将demo模块的zhkf.security.browser.loginType=REDIR

注释掉，不然会被自定义路径拦截掉。

Core模块的validate.code文件下新建ValidateCodeFilter并继承OncePerRequestFilter实现doFilterInternal方法：

**private** AuthenticationFailureHandler authenticationFailureHandler;  
  
**private** SessionStrategy sessionStrategy=**new** HttpSessionSessionStrategy();  
  
@Override  
**protected void** doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, FilterChain filterChain) **throws ServletException**, **IOException** {  
  
 **if**(**StringUtils**.equals("/authentication/form", httpServletRequest.getRequestURI())  
 && **StringUtils**.equalsIgnoreCase(httpServletRequest.getMethod(),"post")){  
 **try**{  
 validate(**new** ServletWebRequest(httpServletRequest));  
 }**catch** (**ValidateCodeException** e){  
 authenticationFailureHandler.onAuthenticationFailure(httpServletRequest,httpServletResponse,e);  
 **return**;  
 }  
  
 }  
 filterChain.doFilter(httpServletRequest,httpServletResponse);  
}  
  
//校验r  
**private void** validate(**ServletWebRequest** request) **throws ServletRequestBindingException** {  
 **ImageCode** codeInSession = (**ImageCode**) sessionStrategy.getAttribute(request,  
 **ValidateCodeController**.SESSION\_KEY);  
  
 **String** codeInRequest = ServletRequestUtils.getStringParameter(request.getRequest(), "imageCode");  
  
 **if** (**StringUtils**.isBlank(codeInRequest)) {  
 **throw new** ValidateCodeException("验证码的值不能为空");  
 }  
  
 **if**(codeInSession == **null**){  
 **throw new** ValidateCodeException("验证码不存在");  
 }  
  
 **if**(codeInSession.isExpried()){  
 sessionStrategy.removeAttribute(request, **ValidateCodeController**.SESSION\_KEY);  
  
 **throw new** ValidateCodeException("验证码已过期");  
 }  
  
 **if**(!**StringUtils**.equals(codeInSession.getCode(), codeInRequest)) {  
 **throw new** ValidateCodeException("验证码不匹配");  
 }  
  
 sessionStrategy.removeAttribute(request, **ValidateCodeController**.SESSION\_KEY);  
}  
  
**public** AuthenticationFailureHandler getAuthenticationFailureHandler() {  
 **return** authenticationFailureHandler;  
}  
  
**public void** setAuthenticationFailureHandler(AuthenticationFailureHandler authenticationFailureHandler) {  
 **this**.authenticationFailureHandler = authenticationFailureHandler;  
}  
  
**public** SessionStrategy getSessionStrategy() {  
 **return** sessionStrategy;  
}  
  
**public void** setSessionStrategy(SessionStrategy sessionStrategy) {  
 **this**.sessionStrategy = sessionStrategy;  
}

新建ValidateCodeException并继承AuthenticationException

**private static final long** serialVersionUID = -2224983836637642756L;  
  
**public** ValidateCodeException(**String** msg){  
 **super**(msg);  
}

ZhkfAuthenticationFailureHandler修改：

httpServletResponse.getWriter().write(objectMapper.writeValueAsString(**new** SimpleResponse(e.getMessage())));

BrowserSecurityConfig.java配置在表单登录之前进行验证码过滤：

**ValidateCodeFilter** validateCodeFilter = **new** ValidateCodeFilter();  
validateCodeFilter.setAuthenticationFailureHandler(zhkfAuthenticationFailureHandler);  
 //表单登录  
http.addFilterBefore(validateCodeFilter, **UsernamePasswordAuthenticationFilter**.**class**)  
 .formLogin()

###### 3.重构代码

(1).验证码基本参数可配置

配置值在调用接口时传递

配置值写在zhkf-security-demo

配置值写在zh-security-core

请求级配置

应用级配置

默认配置

Core模块的propertiies中创建基本属性实体类ImageCodeProperties：

**private int** width = 67;  
**private int** height = 23;  
**private int** length = 4;  
**private int** expireIn = 60;

set\get省略

在创建实体类的对象封装ValidateCodeProperties

**private ImageCodeProperties** image=**new** ImageCodeProperties();

set、get省略

SecurityProperties中初始化ValiDateCodeProperties

**private ValidateCodeProperties** code=**new** ValidateCodeProperties();

set、get省略

ValidateCodeControloler中注入SecurityProperties

@Autowired  
**private SecurityProperties** securityProperties;

createImageCode中修改height与width

**int** width = ServletRequestUtils.getIntParameter(request.getRequest(),"width",securityProperties.getCode().getImage().getWidth());  
**int** height = ServletRequestUtils.getIntParameter(request.getRequest(),"height",securityProperties.getCode().getImage().getHeight());

图片长度修改

**for** (**int** i = 0; i < securityProperties.getCode().getImage().getLength(); i++) {

返回的码数修改

**return new** ImageCode(image, sRand, securityProperties.getCode().getImage().getExpireIn());

(2).验证码拦截的接口可配置

demo配置文件添加：

**zhkf.security.code.image.url** = /user/\*

ImageCodeProperties添加url属性

ValidateCode添加：

**private SecurityProperties** securityProperties;

**private** Set<**String**> urls=**new** HashSet<>();  
**private AntPathMatcher** antPathMatcher=**new** AntPathMatcher();

实现InitializingBean接口，并从写afterPropertiesSet()方法：

@Override  
**public void** afterPropertiesSet() **throws ServletException** {  
 **super**.afterPropertiesSet();  
 **String**[] configUrls=**StringUtils**.splitByWholeSeparatorPreserveAllTokens(securityProperties.getCode().getImage().getUrl(),",");  
 **for**(**String** configUrl:configUrls){  
 urls.add(configUrl);  
 }  
 urls.add("/authentication/form");  
}

doFilterInternal():

**boolean** action = **false**;  
 **for**(**String** url:urls){  
 **if**(antPathMatcher.match(url,httpServletRequest.getRequestURI())){  
 action=**true**;  
 }  
 }  
  
 **if**(action){  
 // if(StringUtils.equals("/authentication/form", httpServletRequest.getRequestURI())  
// && StringUtils.equalsIgnoreCase(httpServletRequest.getMethod(),"post")){  
 **try**{  
 validate(**new** ServletWebRequest(httpServletRequest));  
 }**catch** (**ValidateCodeException** e){  
 authenticationFailureHandler.onAuthenticationFailure(httpServletRequest,httpServletResponse,e);  
 **return**;  
 }  
  
 }  
 filterChain.doFilter(httpServletRequest,httpServletResponse);

BrowserSecurityConfig.java的configure()方法中添加：

validateCodeFilter.setSecurityProperties(securityProperties);  
validateCodeFilter.afterPropertiesSet();

(3).验证码的生成逻辑可配置

(1)在validate.code下创建一个ValidateCodeGenerator接口

**ImageCode** createImageCode(**ServletWebRequest** request);

(2).创建实现方法ImageCodeGenerator实现ValidateCodeGenerator

将ValidateCodeController中的逻辑处理放在ImageCodeGenerator中去，ValidateCodeController中只保留createCode方法。

(3).将ImageCodeGenerator注入到ValidateCodeController中

(4).新建ValidateCodeBeanConfig

@Configuration  
**public class ValidateCodeBeanConfig** {  
 @Autowired  
 **private SecurityProperties** securityProperties;  
 @Bean  
 @ConditionalOnMissingBean(name="imageCodeGenerator")  
 **public** ValidateCodeGenerator imageCodeGenerator(){  
 **ImageCodeGenerator** codeGenerator = **new** ImageCodeGenerator();  
 codeGenerator.setSecurityProperties(securityProperties);  
 **return** codeGenerator;  
 }

@ConditionalOnMissingBean(name="imageCodeGenerator")当启动项目时，程序就会自动上下文扫描imageCodeGenerator接口。

(5).demo模块中新建code.DemoImageCodeGenerator实现ValidateCodeGenerator接口

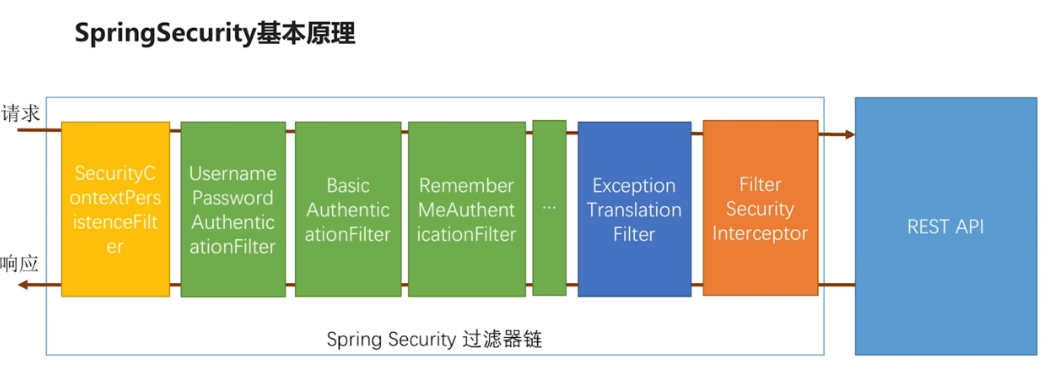
@Component("imageCodeGenerator")  
**public class DemoImageCodeGenerator implements** ValidateCodeGenerator {  
 @Override  
 **public ImageCode** createImageCode(**ServletWebRequest** request) {  
 **System**.out.println("更高级的图形验证码生成代码");  
 **return null**;  
 }  
}

当运行程序启动后，就会进入该自定义方法中，程序启动就会报一个空指针错误。

##### 七、记住我功能

###### 1.基本原理





###### 2.记住我功能具体实现

Html页面新加：

<**tr**>  
 <**td** colspan="2"><**input** name="remember-me" type="checkbox" value="true"/>记住我</**td**>  
</**tr**>

先将自定义的图形验证码配置注释掉。

在BrowserSecurityConfig中配置：

@Autowired  
**private** DataSource dataSource;

@Autowired  
**private** UserDetailsService userDetailsService;

@Bean  
**public** PersistentTokenRepository persistentTokenRepository(){  
 **JdbcTokenRepositoryImpl** tokenRepository = **new** JdbcTokenRepositoryImpl();  
 tokenRepository.setDataSource(dataSource);  
 tokenRepository.setCreateTableOnStartup(**true**); //启动的时候就创建表  
 **return** tokenRepository;  
}

过滤链配置添加：

.and()  
.rememberMe()  
.tokenRepository(persistentTokenRepository())  
//过期的秒数  
.tokenValiditySeconds(securityProperties.getBrowser().getRememberMeSeconds())  
.userDetailsService(userDetailsService)

###### 3.记住我功能SpringSecurity源码解析