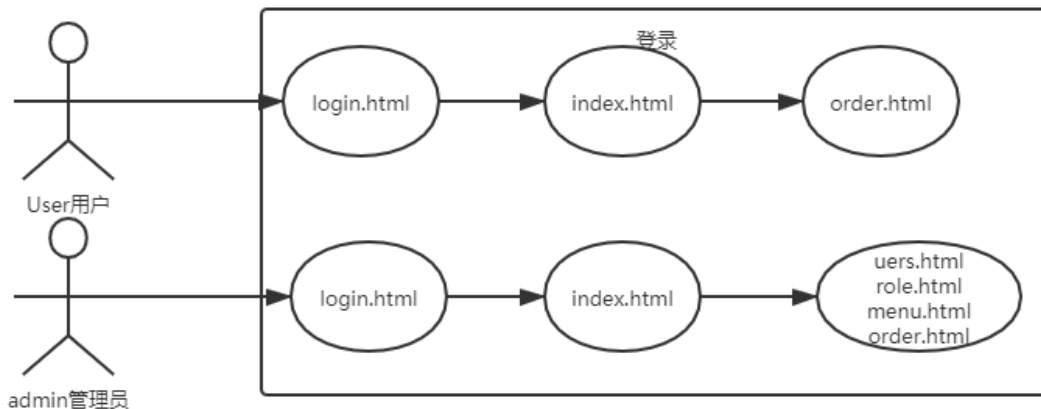


实训邦出品<http://sxbang.net>

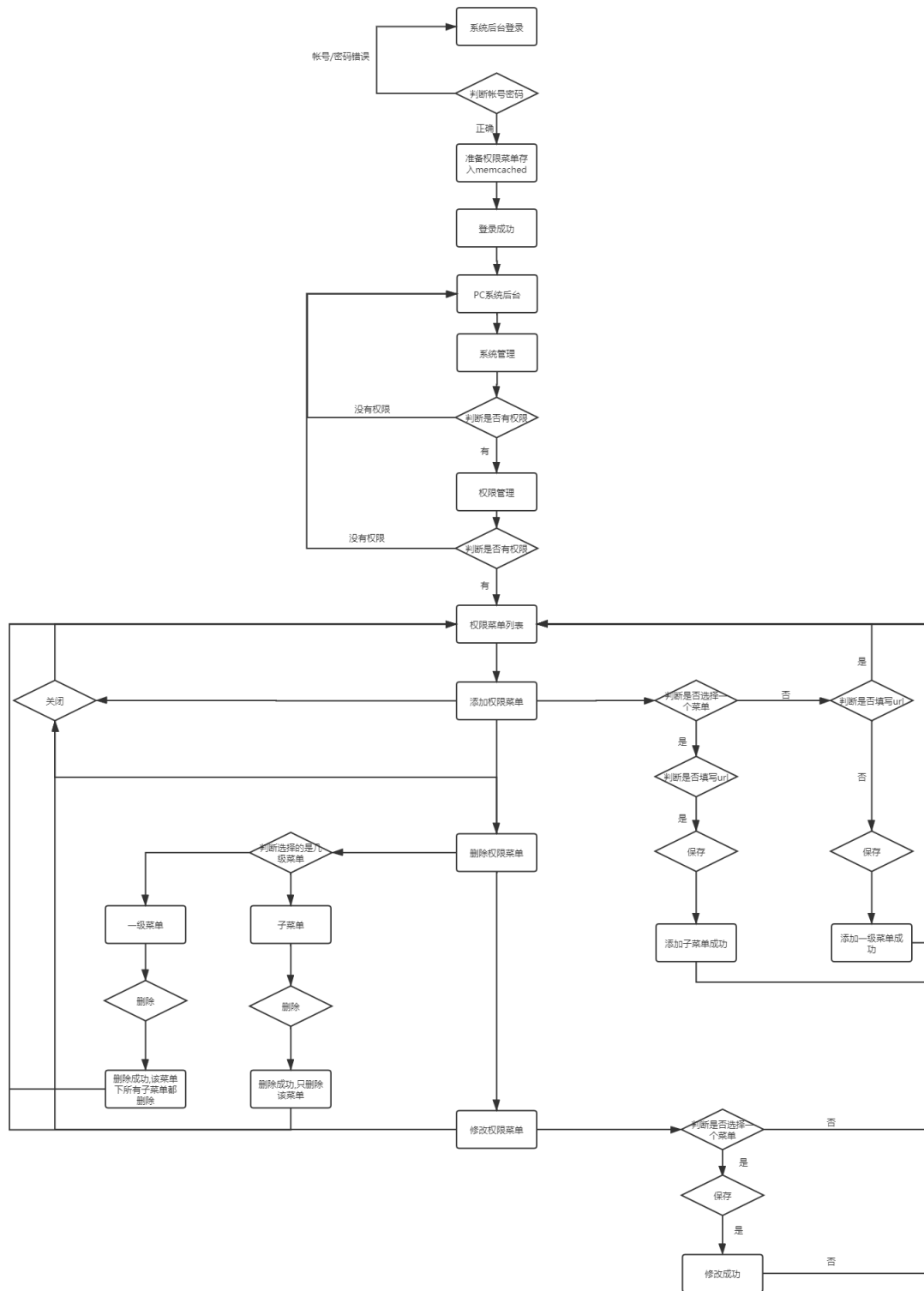
基于Spring boot + Spring Security实现第一版传统架构

任务案例分析

需求用例图



权限管理流程



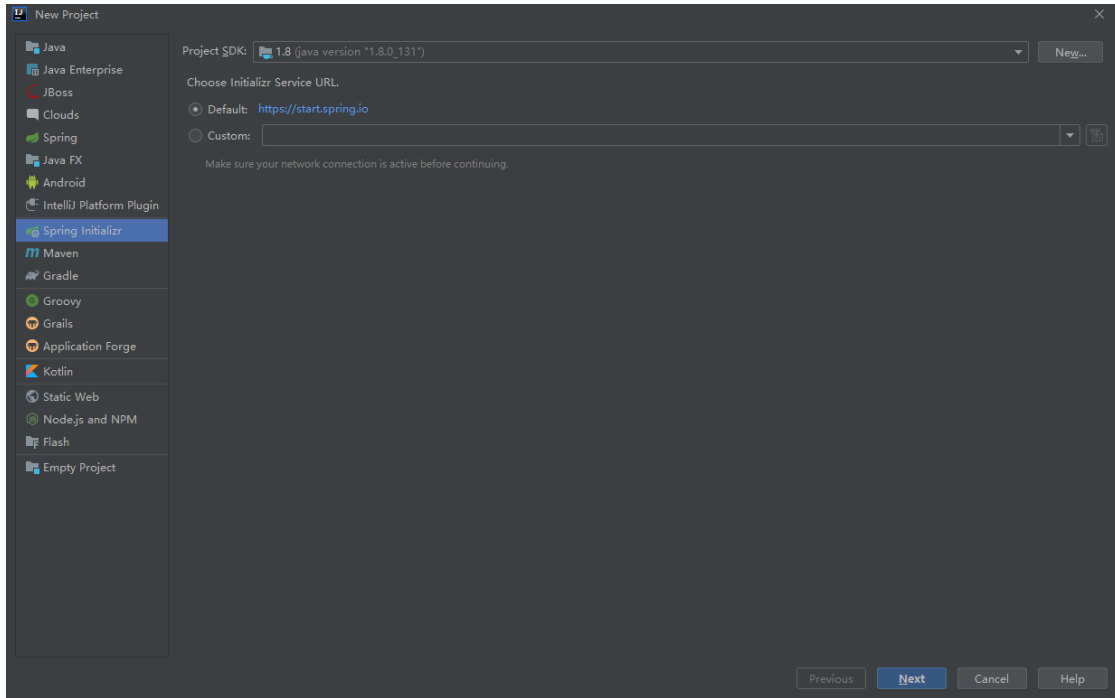
讲义步骤

- 1.使用Spring Security的HttpBasic模式实现登录认证;
- 2.使用Spring Security的FormLogin模式实现登录认证;
- 3.基于JSON的前后端分离开发的登录认证;
- 4.将权限管理系统部署到阿里云的docker;
- 5.基于MySQL数据库的认证和授权。

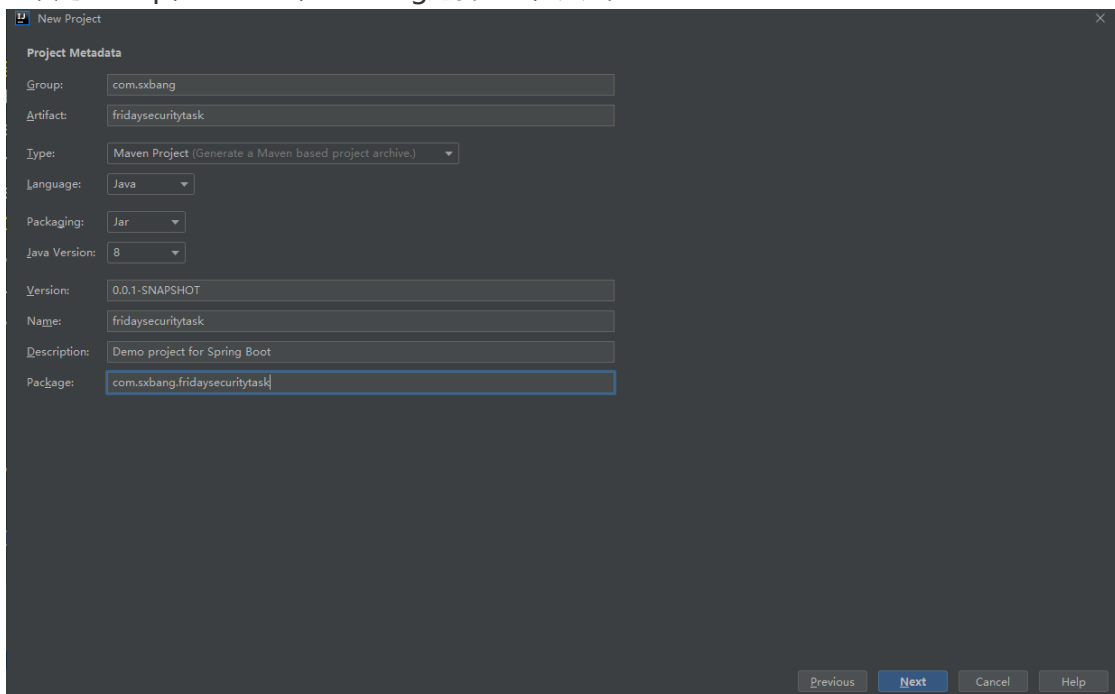
1.使用Spring Security的HttpBasic模式实现登录认证

1.使用Spring Initializr快速构建项目

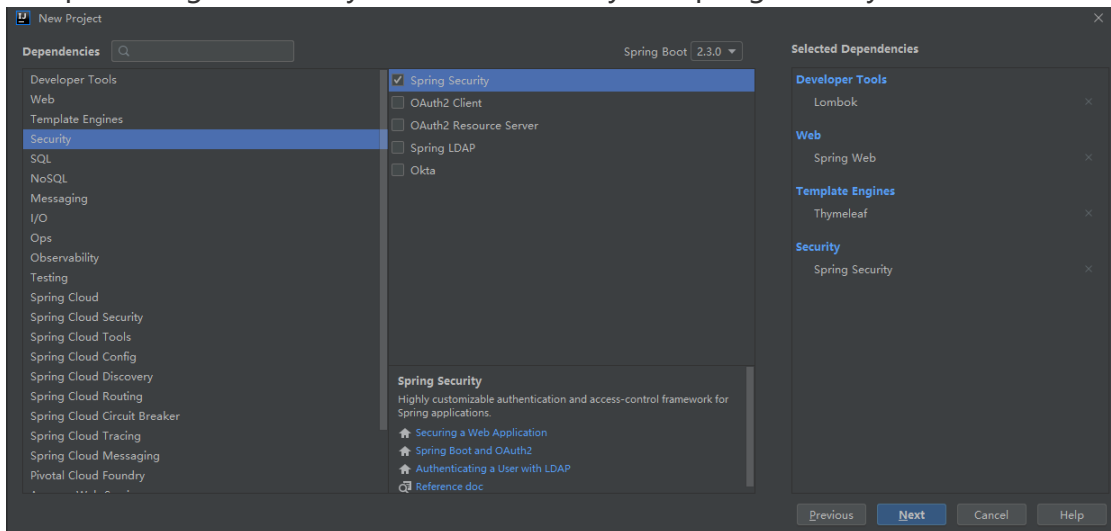
具体步骤： 在IntelliJ IDEA中选择 File -> New -> Project -> Spring Initializr -> 点击Next。



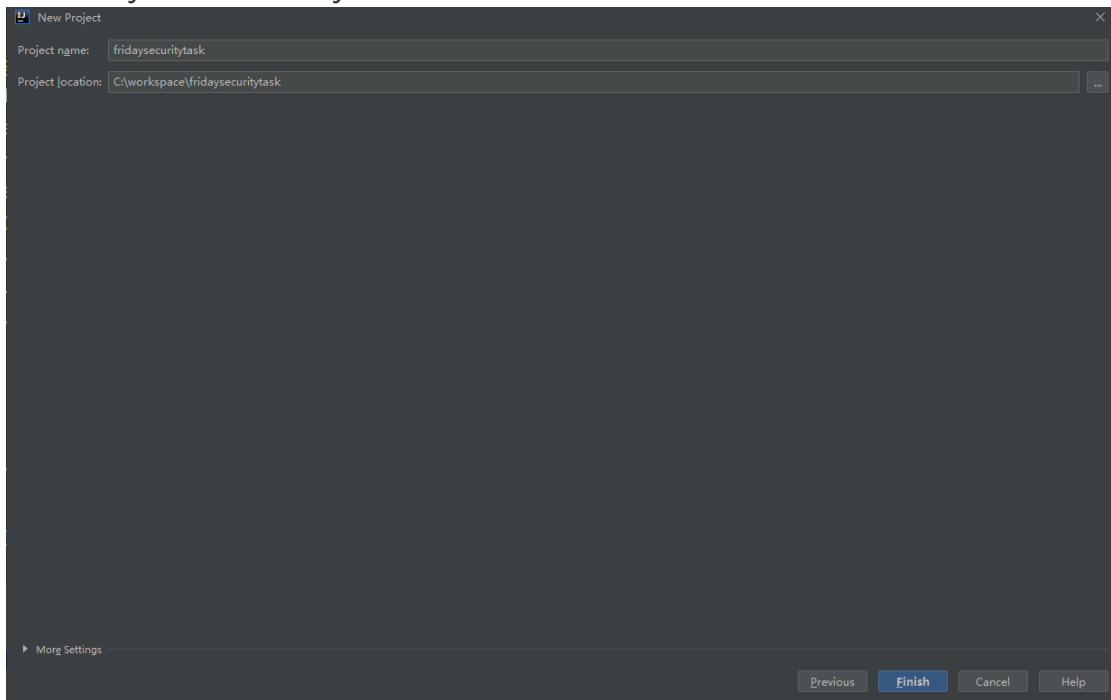
2.填写Group, Artifact, Packing选择Jar，点击Next。



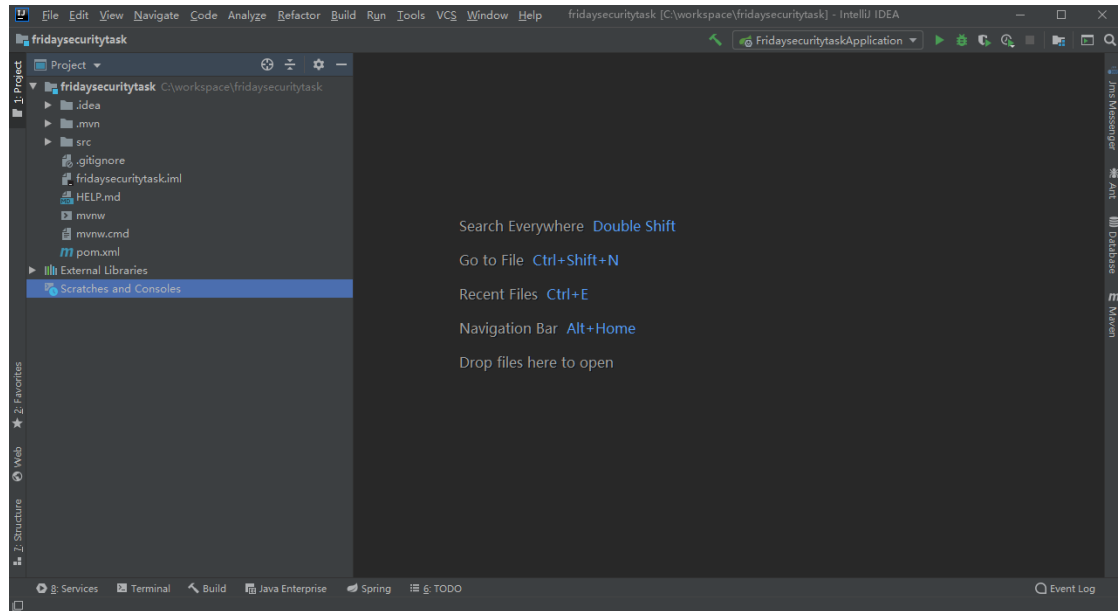
3. 选取依赖，这里我们选择Developer Tools中的Lombok、Web依赖中的Spring Web、Templates Engines中的Thymeleaf 以及Security中的Spring Security，点击Next



4. 核对Project name和Project location，默认不变，选择Finish。

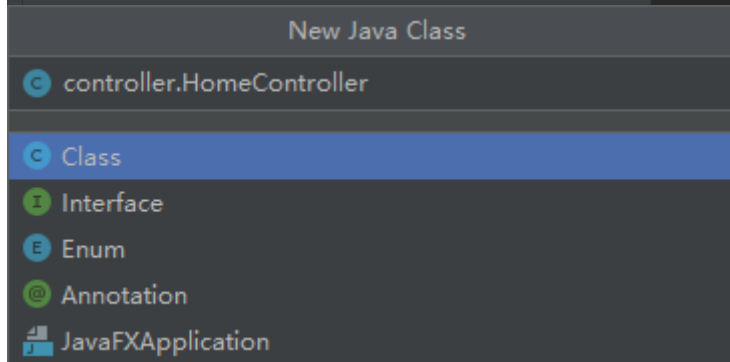
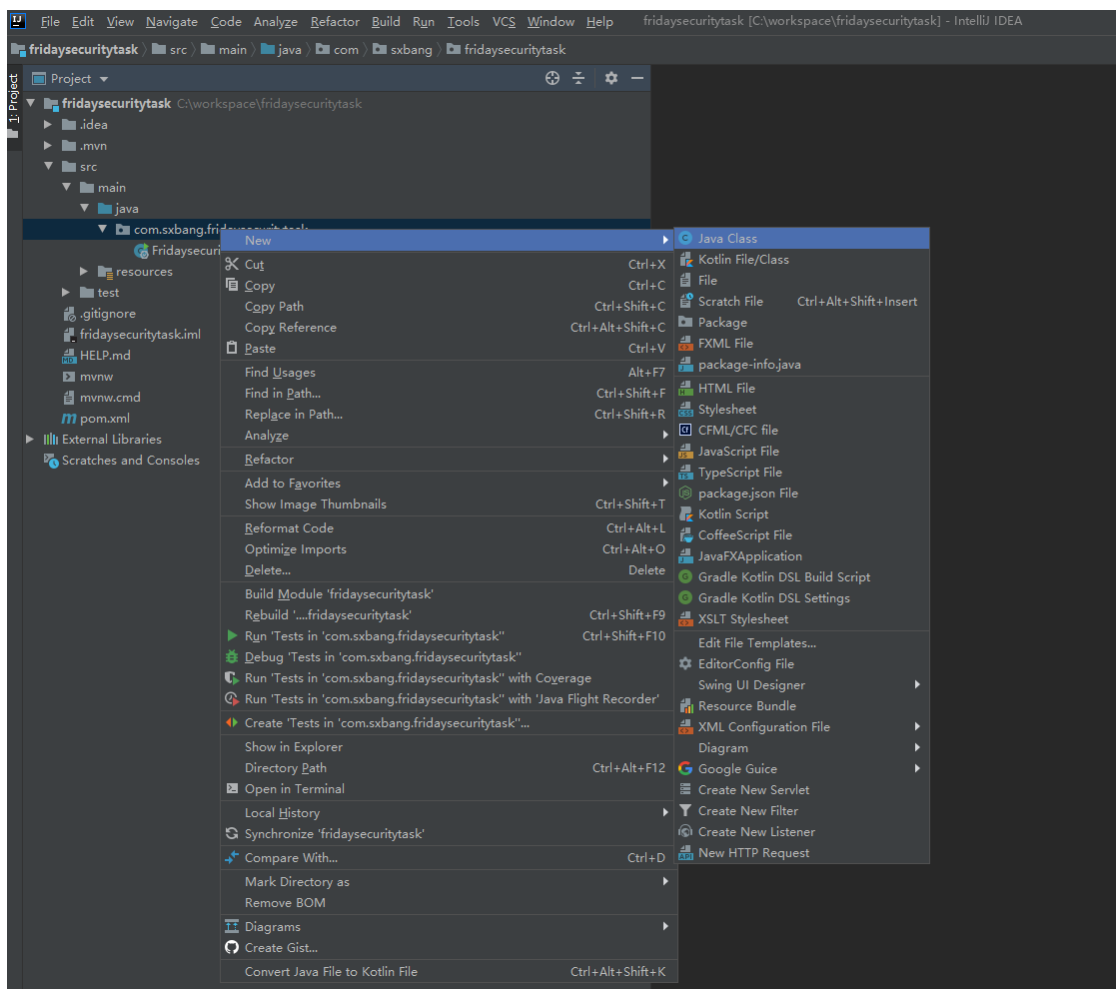


5.构建成功

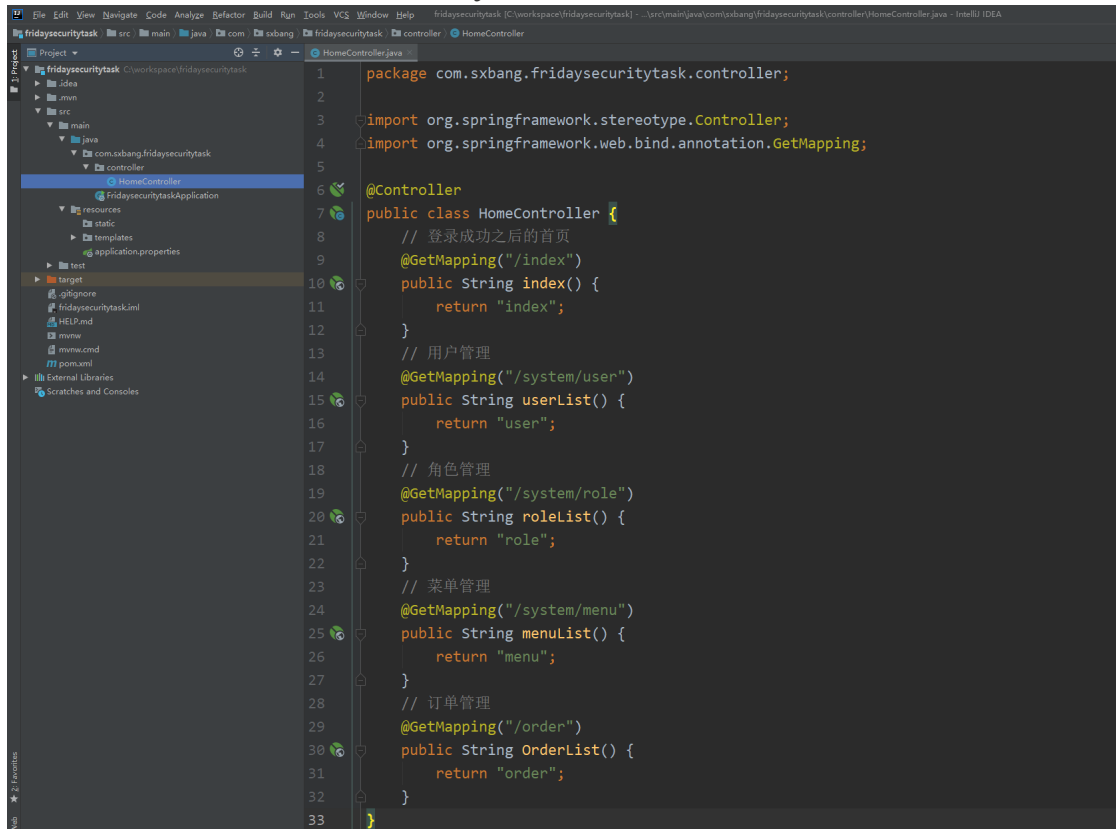


2.使用Thymeleaf制作项目业务页面

具体步骤： 在com.sxbang.fridaysecuritytask包上右键选择 New -> Java Class，输入名字 'controller.HomeController'，点击回车确定。



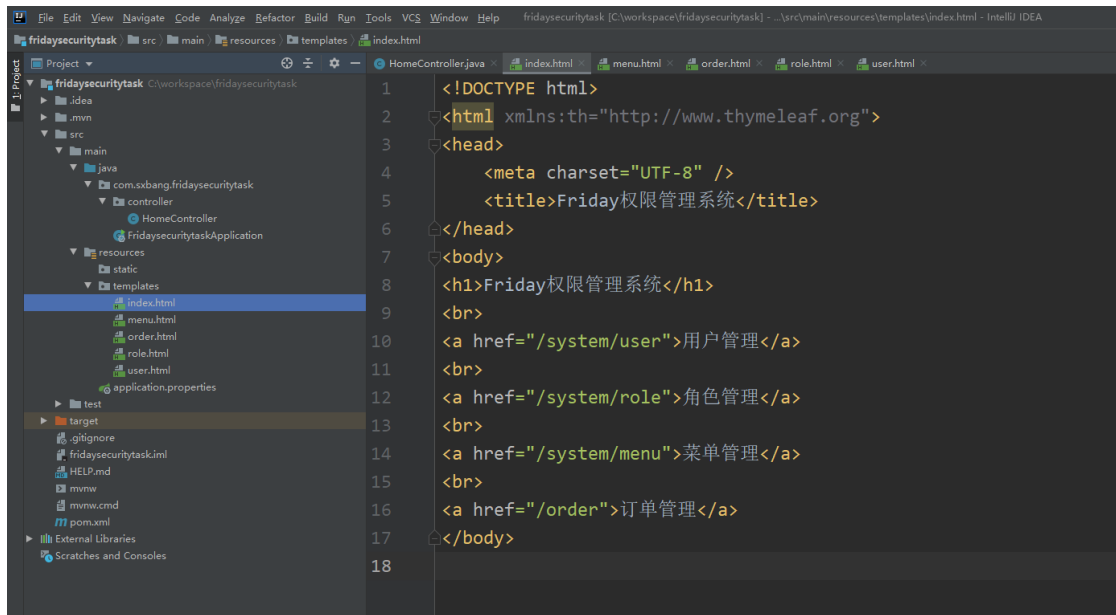
2.根据业务需求编辑HomeController.java。



```
1 package com.sxbang.fridaysecuritytask.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5
6 @Controller
7 public class HomeController {
8     // 登录成功之后的首页
9     @GetMapping("/index")
10    public String index() {
11        return "index";
12    }
13    // 用户管理
14    @GetMapping("/system/user")
15    public String userList() {
16        return "user";
17    }
18    // 角色管理
19    @GetMapping("/system/role")
20    public String roleList() {
21        return "role";
22    }
23    // 菜单管理
24    @GetMapping("/system/menu")
25    public String menuList() {
26        return "menu";
27    }
28    // 订单管理
29    @GetMapping("/order")
30    public String OrderList() {
31        return "order";
32    }
33 }
```

3.使用Thymeleaf制作下面页面：

- index.html



```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Friday权限管理系统</title>
6 </head>
7 <body>
8     <h1>Friday权限管理系统</h1>
9     <br>
10    <a href="/system/user">用户管理</a>
11    <br>
12    <a href="/system/role">角色管理</a>
13    <br>
14    <a href="/system/menu">菜单管理</a>
15    <br>
16    <a href="/order">订单管理</a>
17 </body>
18
```

- user.html

The screenshot shows the IntelliJ IDEA interface with the 'fridaysecuritytask' project open. The 'Project' view on the left shows the file structure, with 'user.html' selected under 'resources/templates'. The 'Editor' view on the right displays the content of 'user.html'.

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Friday权限管理系统</title>
6 </head>
7 <body>
8     <h2>用户列表</h2>
9 <div>
10     <h1>用户页面</h1>
11 </div>
12 </body>
13
```

- role.html

The screenshot shows the IntelliJ IDEA interface with the 'fridaysecuritytask' project open. The 'Project' view on the left shows the file structure, with 'role.html' selected under 'resources/templates'. The 'Editor' view on the right displays the content of 'role.html'.

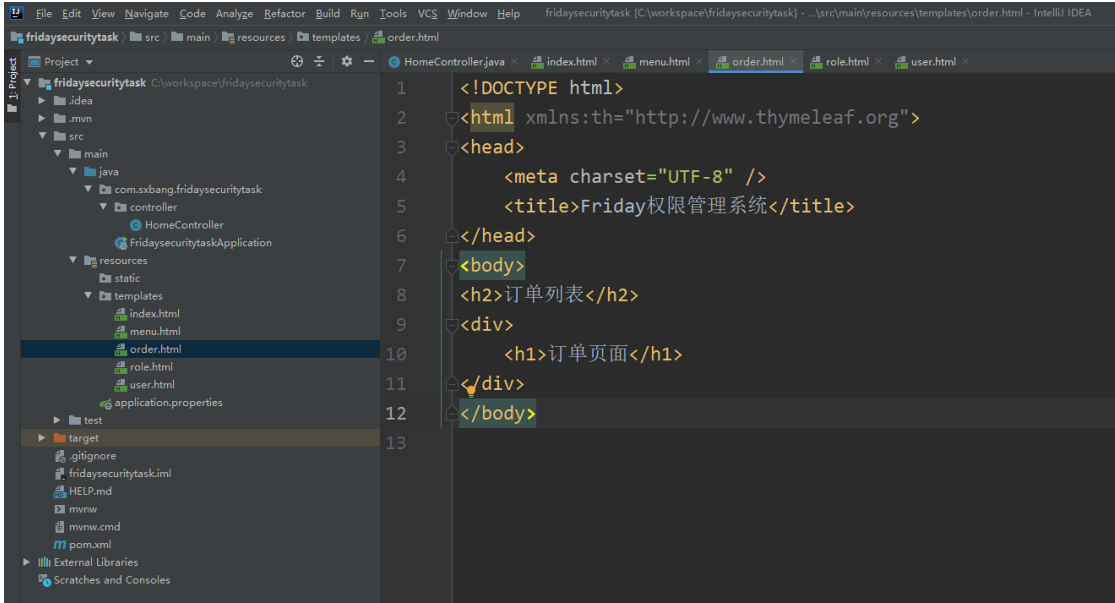
```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Friday权限管理系统</title>
6 </head>
7 <body>
8     <h2>订单列表</h2>
9 <div>
10     <h1>订单页面</h1>
11 </div>
12 </body>
13
```

- menu.html

The screenshot shows the IntelliJ IDEA interface with the 'fridaysecuritytask' project open. The 'Project' view on the left shows the file structure, with 'menu.html' selected under 'resources/templates'. The 'Editor' view on the right displays the content of 'menu.html'.

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Friday权限管理系统</title>
6 </head>
7 <body>
8     <h2>菜单列表</h2>
9 <div>
10     <h1>菜单页面</h1>
11 </div>
12 </body>
13
```

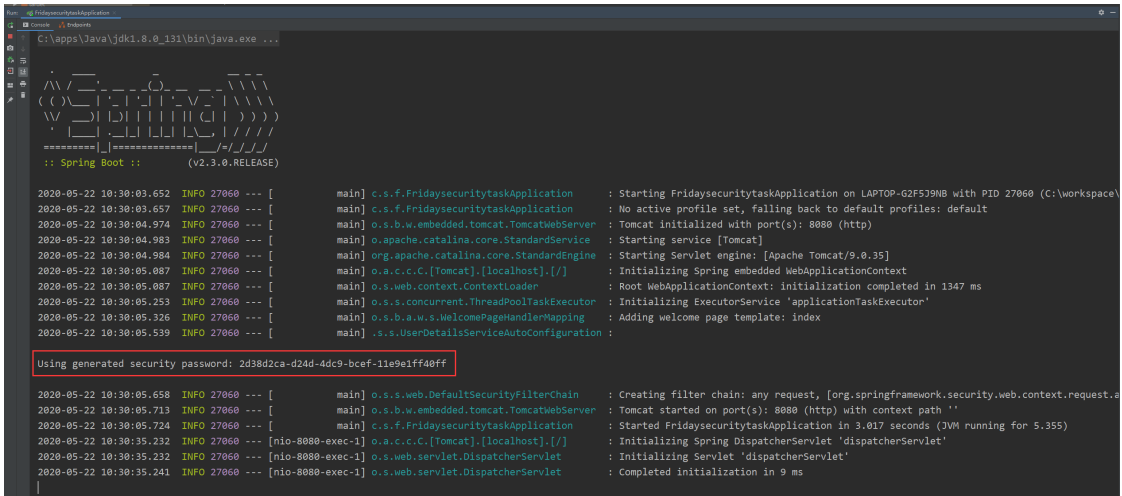

- order.html



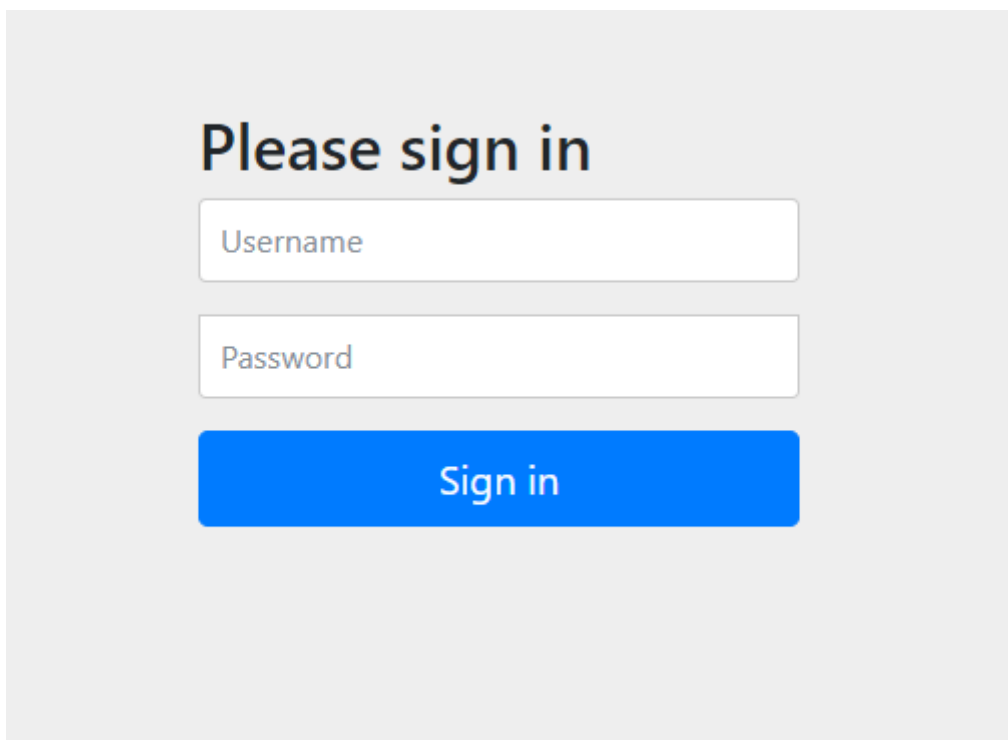
3.启动运行项目，实现Httpbasic模式的登录认证

启动运行项目之后，我们可以看到无需任何配置就实现了登录认证功能，这个就是Spring Security的Httpbasic模式。

1.启动运行项目后，可以在控制台看到输出的密码，我们首先复制这个密码：



2.使用浏览器访问：`http://localhost:8080`，会弹出一个登录框，这个登录框不是我们编码实现的，是由Spring Security来实现。



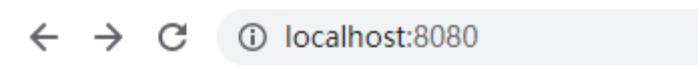
Please sign in

Username

Password

Sign in

3.在登陆页面的Username中输入user，在Password中把刚刚在控制台复制的密码粘贴进去，点击Sign in，就可以成功访问到我们的页面啦。



Friday权限管理系统

[用户管理](#)

[角色管理](#)

[菜单管理](#)

[订单管理](#)

2.使用Spring Security的FormLogin模式实现登录认证

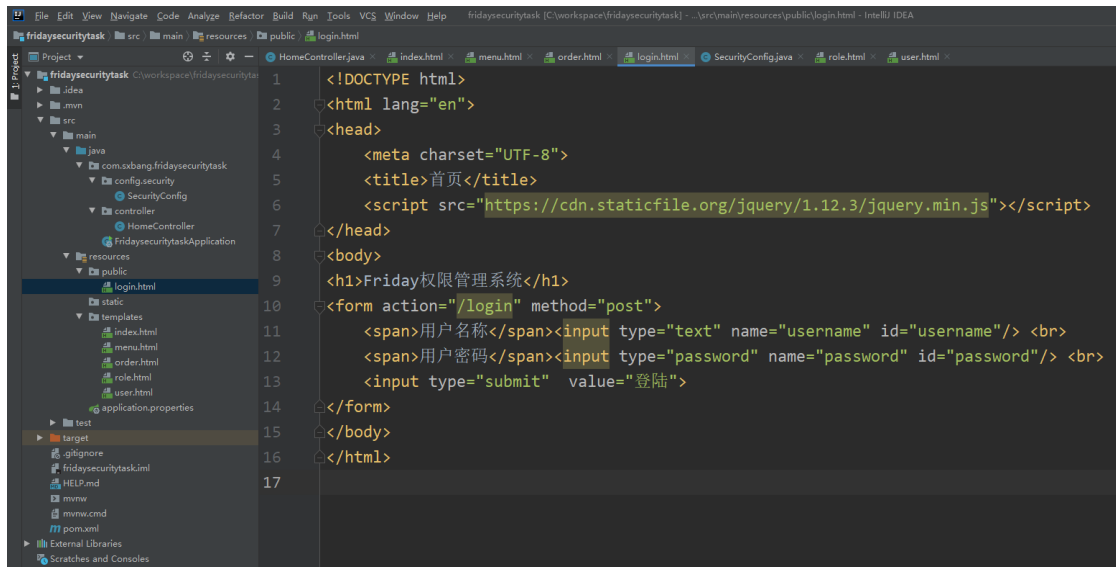
相信大家看过上面HttpBasic模式后发现实际项目应用中它并不适合，因为我们往往都是自己开发一个自定义的登陆页面，Spring Security的FormLogin模式就支持这种需求，下面我们使用FormLogin模式来改写我们的登录认证。

我们先来一起看下需求：

- 1.应用中的所有请求都需要用户登录之后才能访问；

- 2.我们需要自己开发一个登陆页面(login.html);
- 3.我们要允许所有用户有权访问登录页;
- 4.如果用户没有登陆, 必须跳转到登录页进行登录;
- 5.用户成功登陆后, 我们需要根据用户不同的角色进行授权。

下面我们开始使用FormLogin模式, 具体步骤: 1.编写login.html。



```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>首页</title>
6     <script src="https://cdn.staticfile.org/jquery/1.12.3/jquery.min.js"></script>
7 </head>
8 <body>
9     <h1>Friday权限管理系统</h1>
10    <form action="/login" method="post">
11        <span>用户名称</span><input type="text" name="username" id="username"/> <br>
12        <span>用户密码</span><input type="password" name="password" id="password"/> <br>
13        <input type="submit" value="登陆">
14    </form>
15 </body>
16 </html>
17

```

2.创建一个继承WebSecurityConfigurerAdapter的SecurityConfig类, 重写configure(HttpSecurity http) 方法, 用来配置登录验证逻辑。

```

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
            .formLogin()//开启formLogin模式
            .loginPage("/login.html") //用户未登录时, 访问任何资源都跳转到该路径, 即登录页面
            .loginProcessingUrl("/login") //登录表单form中action的地址, 也就是处理认证请求的路径
            .usernameParameter("username") //默认是username
            .passwordParameter("password") //默认是password
            .defaultSuccessUrl("/index") //登录成功跳转接口
            .failureUrl("/login.html") //登录失败跳转页面
            .and() //使用and()连接
            .authorizeRequests() //配置权限
            .antMatchers( ...antPatterns: "/login.html", "/login")
            .permitAll() //用户可以任意访问
            .antMatchers( ...antPatterns: "/order") //需要对外暴露的资源路径
            .hasAnyAuthority( ...authorities: "ROLE_user", "ROLE_admin") //user角色和admin角色都可以访问
            .antMatchers( ...antPatterns: "/system/user", "/system/role", "/system/menu")
            .hasAnyRole( ...roles: "admin") //admin角色可以访问
            // 除上面外的所有请求全部需要鉴权认证
            .anyRequest().authenticated() //authenticated()要求在执行该请求时, 必须已经登录了应用
            .and()
            .csrf().disable() ;//禁用跨站csrf攻击防御, 否则无法登陆成功
    }
}

```

上图代码分三段理解:

1.配置认证, 开启formLogin模式;

- 2.配置权限;
- 3.禁用跨站csrf攻击防御。

```
/**
 * anyRequest          | 匹配所有请求路径
 * access              | SpringEl表达式结果为true时可以访问
 * anonymous            | 匿名可以访问
 * denyAll             | 用户不能访问
 * fullyAuthenticated  | 用户完全认证可以访问（非remember-me下自动登录）
 * hasAnyAuthority      | 如果有参数，参数表示权限，则其中任何一个权限可以访问
 * hasAnyRole           | 如果有参数，参数表示角色，则其中任何一个角色可以访问
 * hasAuthority         | 如果有参数，参数表示权限，则其权限可以访问
 * hasIpAddress         | 如果有参数，参数表示IP地址，如果用户IP和参数匹配，则可以访问
 * hasRole              | 如果有参数，参数表示角色，则其角色可以访问
 * permitAll           | 用户可以任意访问
 * rememberMe           | 允许通过remember-me登录的用户访问
 * authenticated        | 用户登录后可访问
 */
```

3.这里我们采用内存中身份认证的方法，在SecurityConfig类重写configure(AuthenticationManagerBuilder auth)方法，增加user和admin两个用户的配置，后续我们会根据RBAC模型设计数据表，实现基于数据库动态的配置。

官方推荐使用BCryptPasswordEncoder进行密码加密。

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().inMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
        .withUser( username: "user") UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
            .password(bCryptPasswordEncoder().encode( rawPassword: "123456")) UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
            .roles("user") UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
            .and() InMemoryUserDetailsServiceConfigurer<AuthenticationManagerBuilder>
                .withUser( username: "admin") UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
                    .password(bCryptPasswordEncoder().encode( rawPassword: "123456")) UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
                    .roles("admin") UserDetailsServiceConfigurer<B, C>.UserDetailsServiceBuilder
                    .and() InMemoryUserDetailsServiceConfigurer<AuthenticationManagerBuilder>
                        .passwordEncoder(bCryptPasswordEncoder()); //配置BCrypt加密
}

/**
 * 强散列哈希加密实现
 */
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder()
{
    return new BCryptPasswordEncoder();
}
```

4.运行验证，使用浏览器访问：<http://localhost:8080>。

```
C:\apps\java\jdk-8_0_tsl\bin>java.exe ...
```

```
\n / _ _   \n\n(( )) | .| | | | | | | | | | | |\n(W )_|_|_|_|_|_|_|_|_|_|_|_|)|)\n' |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_\n=====|=====|_/~///~/\n\n:: Spring Boot ::      (v2.3.0.RELEASE)
```

```
2020-05-22 16:45:57.637 INFO 18020 --- [main] c.s.f.FridaysecuritytaskApplication : Starting FridaysecuritytaskApplication on LAPTOP-G2F5J9NB with PID 18020 (C:\\workspace\\  
2020-05-22 16:45:57.642 INFO 18020 --- [main] c.s.f.FridaysecuritytaskApplication : No active profile set, falling back to default profiles: default  
2020-05-22 16:45:58.940 INFO 18020 --- [main] o.a.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)  
2020-05-22 16:45:58.950 INFO 18020 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]  
2020-05-22 16:45:58.950 INFO 18020 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.35]  
2020-05-22 16:45:59.072 INFO 18020 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext  
2020-05-22 16:45:59.072 INFO 18020 --- [main] o.s.web.context.ContextLoader       : Root WebApplicationContext: initialization completed in 1356 ms  
2020-05-22 16:45:59.276 INFO 18020 --- [main] o.s.c.c.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'  
2020-05-22 16:45:59.333 INFO 18020 --- [main] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page template: index  
2020-05-22 16:45:59.760 INFO 18020 --- [main] o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: any request, [org.springframework.security.web.context.request.  
2020-05-22 16:45:59.856 INFO 18020 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''  
2020-05-22 16:45:59.875 INFO 18020 --- [main] c.s.f.FridaysecuritytaskApplication : main    c.s.f.FridaysecuritytaskApplication in 2.841 seconds (JVM running for 4.653)
```

Friday权限管理系统

用户名称	<input type="text"/>
用户密码	<input type="password"/>
登陆	<input type="button" value="登陆"/>

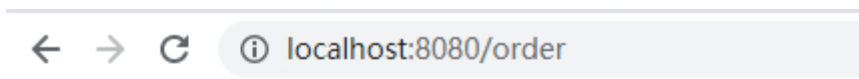
在username中输入user，password输入123456，点击登录按钮，我们可以成功登录到系统中。

Friday权限管理系统

- 用户管理
- 角色管理
- 菜单管理
- 订单管理

根据权限配置，user用户可以访问订单页面，不能访问用户管理、角色管理和菜单管理，下面我们分别访问订单页面和用户管理页面，看一下是不是和我们的代码配置一致。

- 点击订单页面，可以正常访问：



订单列表

订单页面

- 点击用户管理页面，提示我们被禁止访问：

← → ↺ ⓘ localhost:8080/system/user

Whitelabel Error Page

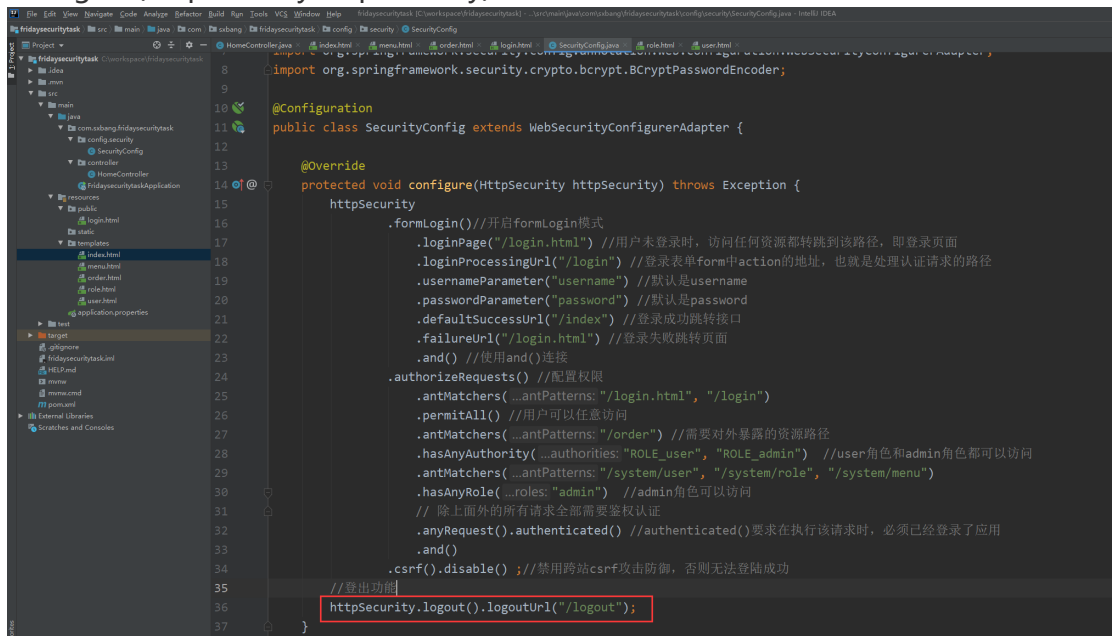
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri May 22 17:05:21 CST 2020

There was an unexpected error (type=Forbidden, status=403).

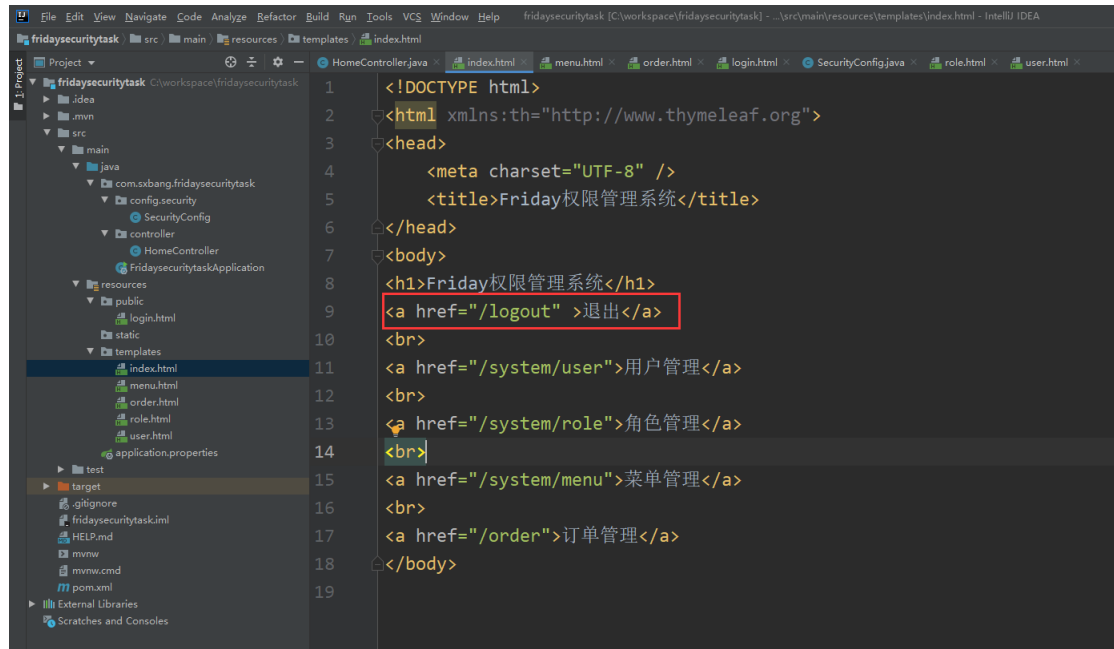
5.一行代码实现登出功能。

Spring Security帮我们实现登出功能的大部分代码，我们只需要在configure(HttpSecurity httpSecurity)方法内添加一行即可：



```
8      import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
9
10     @Configuration
11     public class SecurityConfig extends WebSecurityConfigurerAdapter {
12
13         @Override
14         protected void configure(HttpSecurity httpSecurity) throws Exception {
15             httpSecurity
16                 .formLogin() // 开启formLogin模式
17                 .loginPage("/login.html") // 用户未登录时，访问任何资源都跳转到该路径，即登录页面
18                 .loginProcessingUrl("/login") // 登录表单form中action的地址，也就是处理认证请求的路径
19                 .usernameParameter("username") // 默认是username
20                 .passwordParameter("password") // 默认是password
21                 .defaultSuccessUrl("/index") // 登录成功跳转接口
22                 .failureUrl("/login.html") // 登录失败跳转页面
23                 .and() // 使用and()连接
24                 .authorizeRequests() // 配置权限
25                 .antMatchers( ... antPatterns: "/login.html", "/login")
26                 .permitAll() // 用户可以任意访问
27                 .antMatchers( ... antPatterns: "/order") // 需要对外暴露的资源路径
28                 .hasAnyAuthority( ... authorities: "ROLE_user", "ROLE_admin") // user角色和admin角色都可以访问
29                 .antMatchers( ... antPatterns: "/system/user", "/system/role", "/system/menu")
30                 .hasAnyRole( ... roles: "admin") // admin角色可以访问
31                 // 除上面外的所有请求全部需要鉴权认证
32                 .anyRequest().authenticated() // authenticated() 要求在执行该请求时，必须已经登录了应用
33                 .and()
34                 .csrf().disable(); // 禁用跨站csrf攻击防御，否则无法登陆成功
35
36                 // 登出功能
37                 httpSecurity.logout().logoutUrl("/logout");
38         }
39     }
```

然后在index.html中增加 ‘<a >退出’ 即可。



```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Friday权限管理系统</title>
6 </head>
7 <body>
8 <h1>Friday权限管理系统</h1>
9 <a href="/logout" >退出</a>
10 <br>
11 <a href="/system/user">用户管理</a>
12 <br>
13 <a href="/system/role">角色管理</a>
14 <br>
15 <a href="/system/menu">菜单管理</a>
16 <br>
17 <a href="/order">订单管理</a>
18 </body>
19
```

3.基于JSON的前后端分离开发的登录认证

前面的例子，在发送登录请求并认证成功之后，页面会跳转回原访问页，但在前后端分离开发、通过JSON数据完成交互的应用中，会在登录时返回一段JSON数据，告知前端登录成功与否，由前端决定如何处理后续逻辑，而非由服务器主动执行页面跳转，下面我们就看看这种情况如何实现。

Spring Security表单登录配置模块提供了successHandler ()和failureHandler ()两个方法，分别处理登录成功和登录失败的逻辑。其中，successHandler ()方法带有一个Authentication参数，携带当前登录用户名及其角色等信息；而failureHandler ()方法携带一个AuthenticationException异常参数。具体处理方式需按照系统的情况自定义。

实现思路：

- 1.修改login.html，使用JSON数据传递username和password；
- 2.自定义登陆成功时的处理逻辑；
- 3.自定义登录失败时的处理逻辑。

实现步骤：1.修改login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>首页</title>
  <script src="https://cdn.staticfile.org/jquery/1.12.3/jquery.min.js"></script>
</head>
<body>
<h1>Friday权限管理系统</h1>
<form action="/login" method="post">
  <span>用户名</span><input type="text" name="username" id="username"/> <br>
  <span>用户密码</span><input type="password" name="password" id="password"/> <br>
  <!-- <input type="submit" value="登陆"-->
  <input type="button" onclick="login()" value="登陆">
</form>
</body>
<script>
  function login() {
    $.ajax({
      type: "POST",
      url: "/login",
      data: {
        "username": $("#username").val(),
        "password": $("#password").val()
      },
      success: function (data) {
        if(data.code == 20001){
          location.href = "/index";
        }else{
          alert(data.msg);
        }
      }
    });
  }
</script>
</html>
```

2.在com.sxbang.fridaysecuritytask.config.security.handler包下创建MyAuthenticationSuccessHandler类，使它实现AuthenticationSuccessHandler接口，重写onAuthenticationSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) 方法来实现登陆成功返回逻辑。

```
@Component
public class MyAuthenticationSuccessHandler implements AuthenticationSuccessHandler {
    @Override
    public void onAuthenticationSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication)
        throws IOException, ServletException {
        httpServletResponse.setContentType("application/json;charset=UTF-8");
        PrintWriter out = httpServletResponse.getWriter();
        out.write(s: "{\"code\":\"20001\",\"msg\":\"登陆成功\"}");
    }
}
```

3.在com.sxbang.fridaysecuritytask.config.security.handler包下创建MyAuthenticationFailureHandler类，使它实现AuthenticationFailureHandler接口，重写onAuthenticationFailure(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, AuthenticationException e) 方法来实现

登陆失败返回逻辑。

```
@Component
public class MyAuthenticationFailureHandler implements AuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, AuthenticationException e)
        throws IOException, ServletException {
        httpServletResponse.setContentType("application/json;charset=UTF-8");
        PrintWriter out = httpServletResponse.getWriter();
        out.write(s: "{\"code\":\"40001\",\"msg\":\"登陆失败\"}");
    }
}
```

4.在configure(HttpSecurity httpSecurity)方法中分别调用
successHandler(successHandler)和failureHandler(failureHandler)方法来实现自定义处
理逻辑

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Resource
    private MyAuthenticationSuccessHandler successHandler;

    @Resource
    private MyAuthenticationFailureHandler failureHandler;

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        httpSecurity
            .formLogin()//开启formLogin模式
            .loginPage("/login.html") //用户未登录时，访问任何资源都跳转到该路径，即登录页面
            .loginProcessingUrl("/login") //登录表单form中action的地址，也就是处理认证请求的路径
            .usernameParameter("username") //默认是username
            .passwordParameter("password") //默认是password
            .defaultSuccessUrl("/index") //登录成功跳转接口
            .failureUrl("/login.html") //登录失败跳转页面
            .successHandler(successHandler)
            .failureHandler(failureHandler)
            .and() //使用and()连接
            .authorizeRequests() //配置权限
            .antMatchers( ...antPatterns: "/login.html", "/login")
            .permitAll() //用户可以任意访问
            .antMatchers( ...antPatterns: "/order") //需要对外暴露的资源路径
            .hasAnyAuthority( ...authorities: "ROLE_user", "ROLE_admin") //user角色和admin角色都可以访问
            .antMatchers( ...antPatterns: "/system/user", "/system/role", "/system/menu")
            .hasAnyRole( ...roles: "admin") //admin角色可以访问
            // 除上面外的所有请求全部需要鉴权认证
            .anyRequest().authenticated() //authenticated() 要求在执行该请求时，必须已经登录了应用
            .and()
            .csrf().disable() ;//禁用跨站csrf攻击防御，否则无法登陆成功
    }
}
```

5.运行检验效果

4.将权限管理系统部署到阿里云的 docker

本节内容，我们使用IntelliJ IDEA的Docker插件帮助我们将当前权限管理应用制作成
Docker镜像、运行在指定的远程机器(阿里云)上。

实现步骤：1.在CentSO系统上开启Docker的远程连接，如果你对docker安装和基本的操
作还不熟悉，请参考我的docker课程之后在继续下面的内容。

- 编辑此文件：/lib/systemd/system/docker.service，把
ExecStart=/usr/bin/dockerd-current 改为ExecStart=/usr/bin/dockerd-current
-H tcp://0.0.0.0:2375 -H unix://var/run/docker.sock \，如下：

```

1 [Unit]
2 Description=Docker Application Container Engine
3 Documentation=https://docs.docker.com
4 After=network-online.target firewall.service
5 Wants=network-online.target
6
7 [Service]
8 Type=notify
9 # the default is not to use systemd for cgroups because the delegate issues still
10 # exists and systemd currently does not support the cgroup feature set required
11 # for containers run by docker
12 ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
13 ExecReload=/bin/kill -s HUP $MAINPID
14 # Having non-zero Limit*s causes performance problems due to accounting overhead
15 # in the kernel. We recommend using cgroups to do container-local accounting.
16 LimitNOFILE=infinity
17 LimitNPROC=infinity
18 LimitCORE=infinity
19 # Uncomment TasksMax if your systemd version supports it.
20 # Only systemd 226 and above support this version.
21 #TasksMax=infinity
22 TimeoutStartSec=0
23 # set delegate yes so that systemd does not reset the cgroups of docker containers
24 Delegate=yes
25 # kill only the docker process, not all processes in the cgroup
26 KillMode=process
27 # restart the docker process if it exits prematurely
28 Restart=on-failure
29 StartLimitBurst=3
30 StartLimitInterval=60s
31
32 [Install]
33 WantedBy=multi-user.target

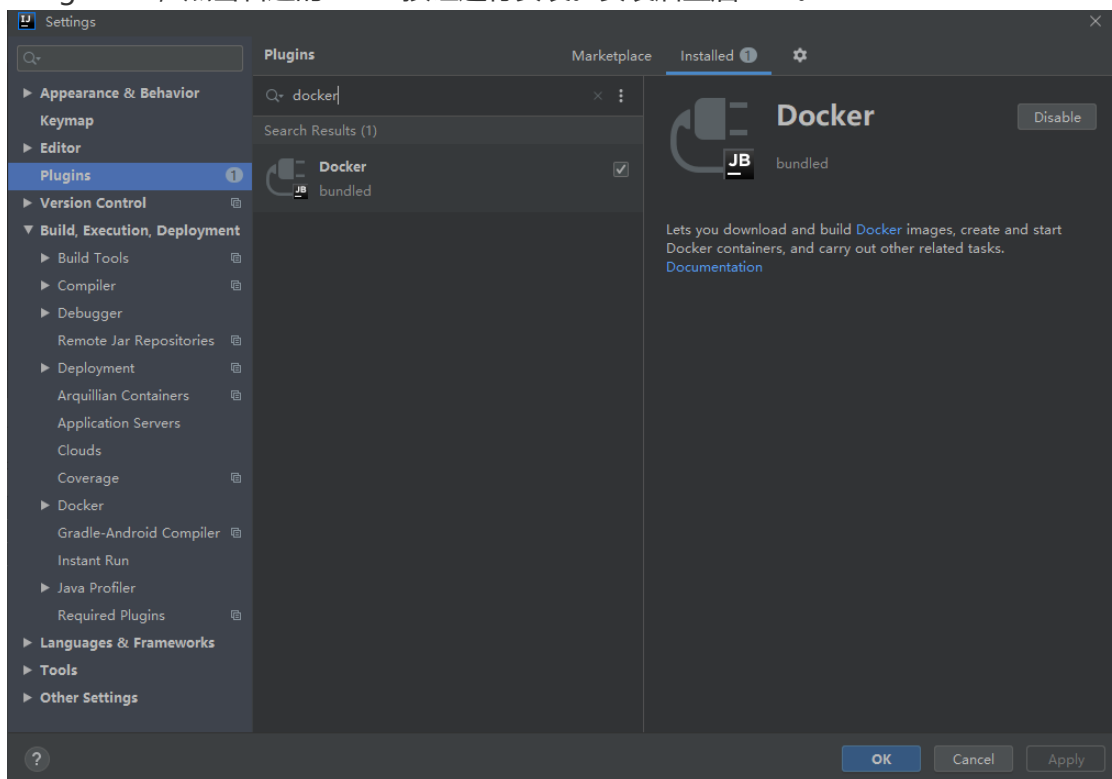
```

- 重新加载配置并重启docker:

```
1 | systemctl daemon-reload && systemctl restart docker
```

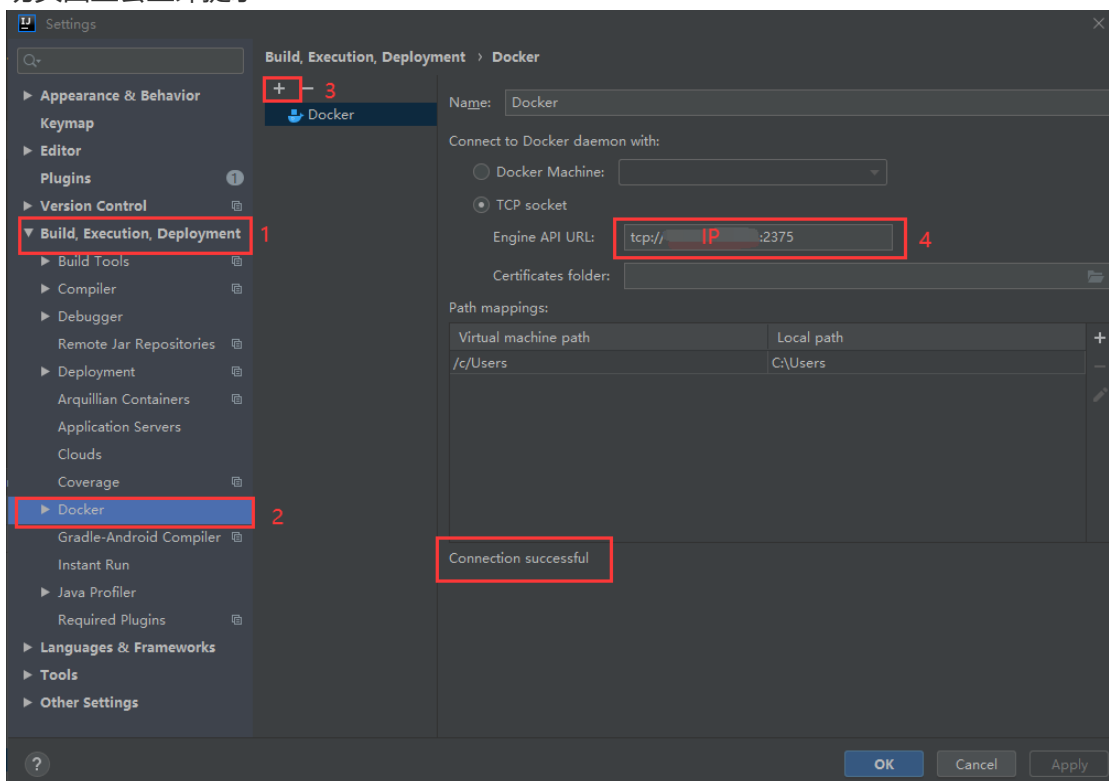
- 确保端口2375已开启，如果使用阿里云等云服务，记得在安全策略上配置端口2375

2.IntelliJ IDEA安装Docker插件，打开Idea，从File->Settings->Plugins->Install JetBrains plugin进入插件安装界面，在搜索框中输入docker，可以看到Docker integration，点击右边的Install按钮进行安装。安装后重启Idea。

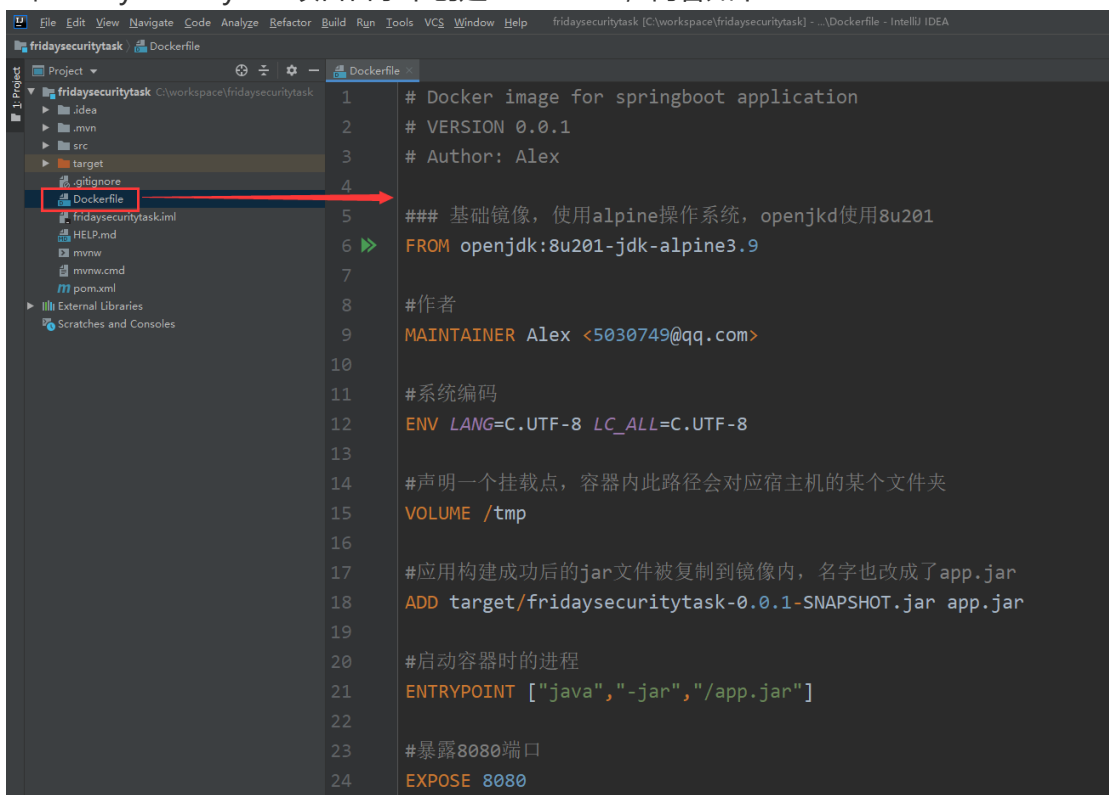


3.重启后配置docker，连接到远程docker服务。从File->Settings->Build,Execution,Deployment->Docker打开配置界面。在设置页面，按照下图的数字顺序创建一个Docker server并进行设置，输入Docker服务所在机器的IP地址，如果连接成

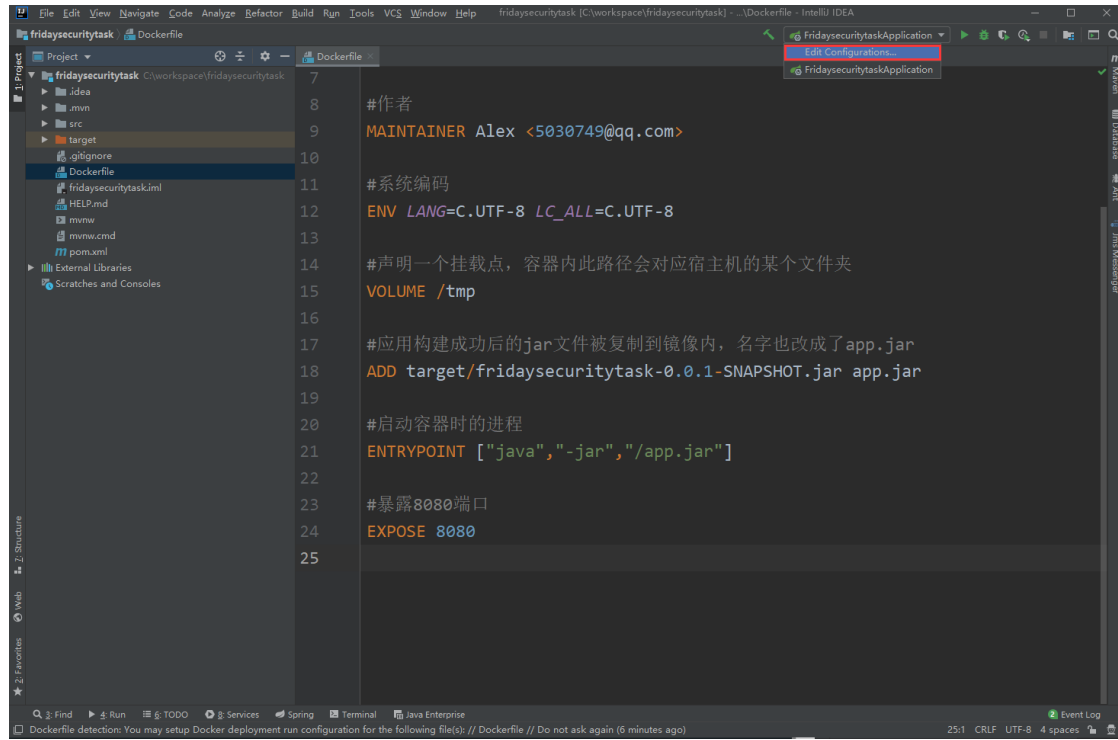
功页面上会立即提示"Connection successful"

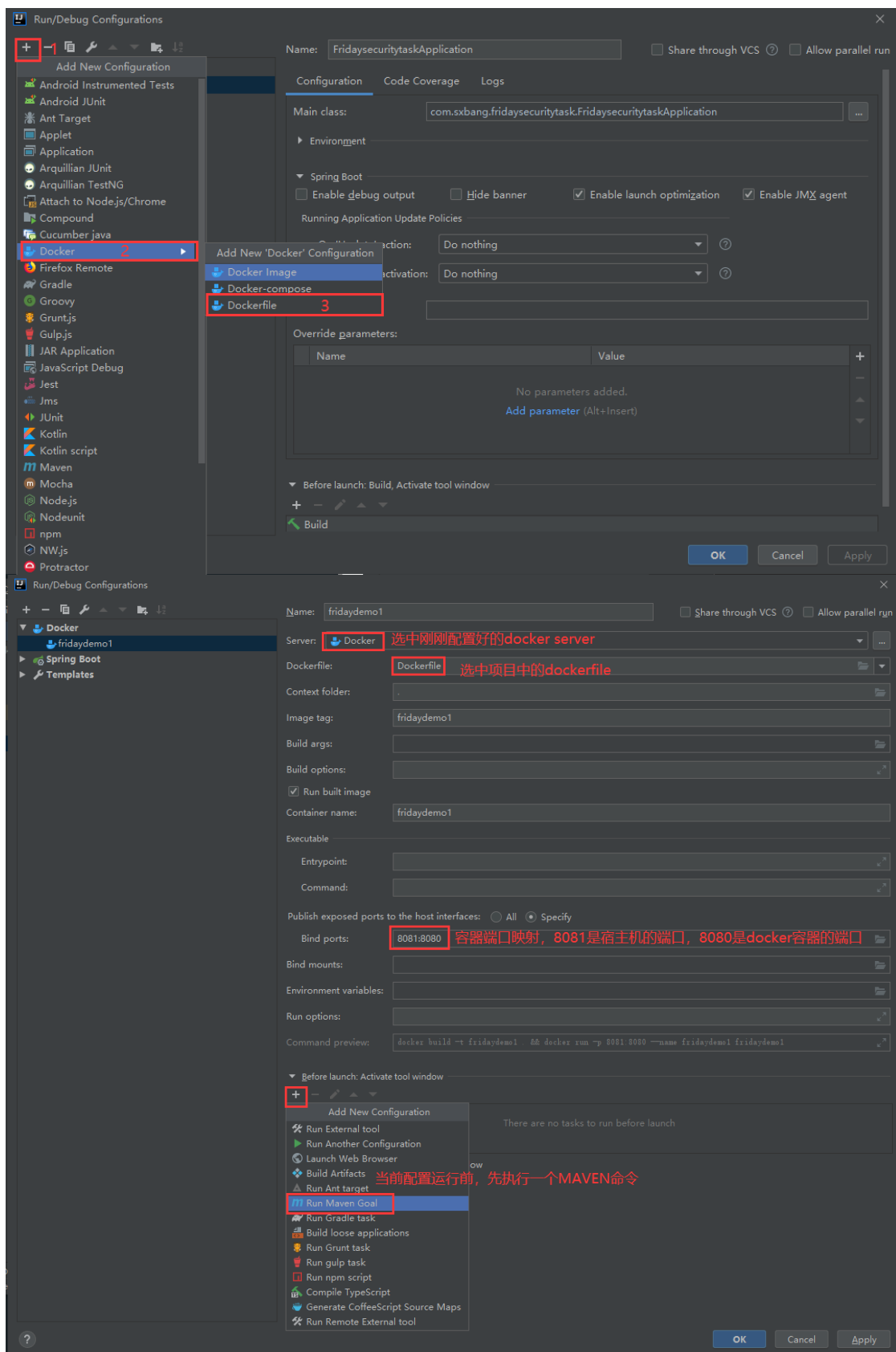


4.在fridaysecuritytask项目目录下创建Dockerfile，内容如下：

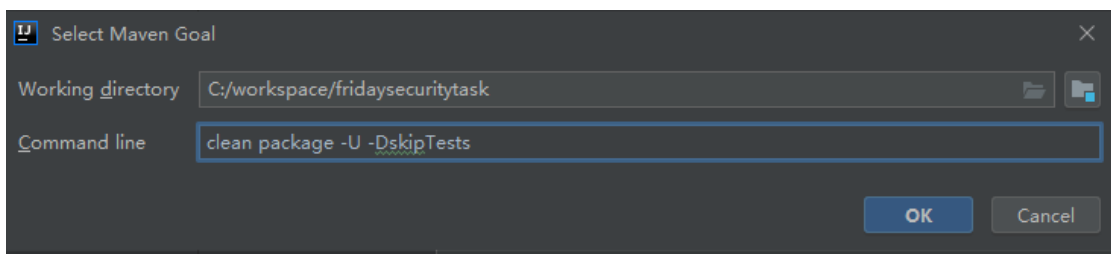


5.按照下图操作，创建一个Dockerfile的配置

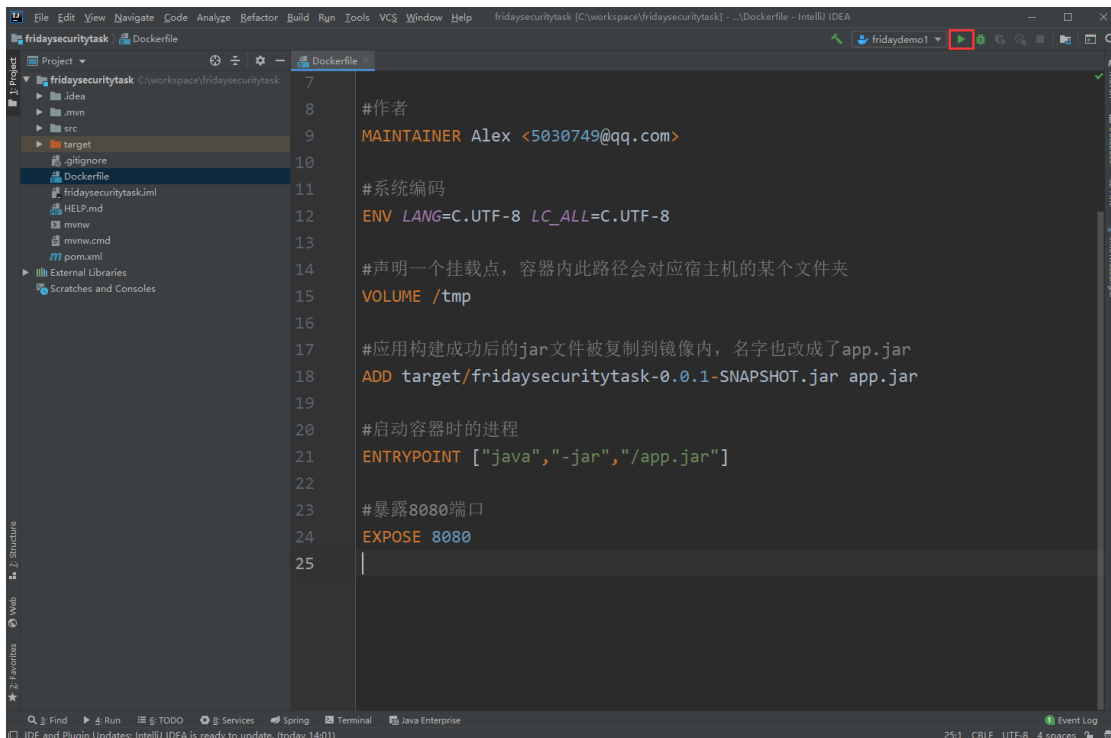




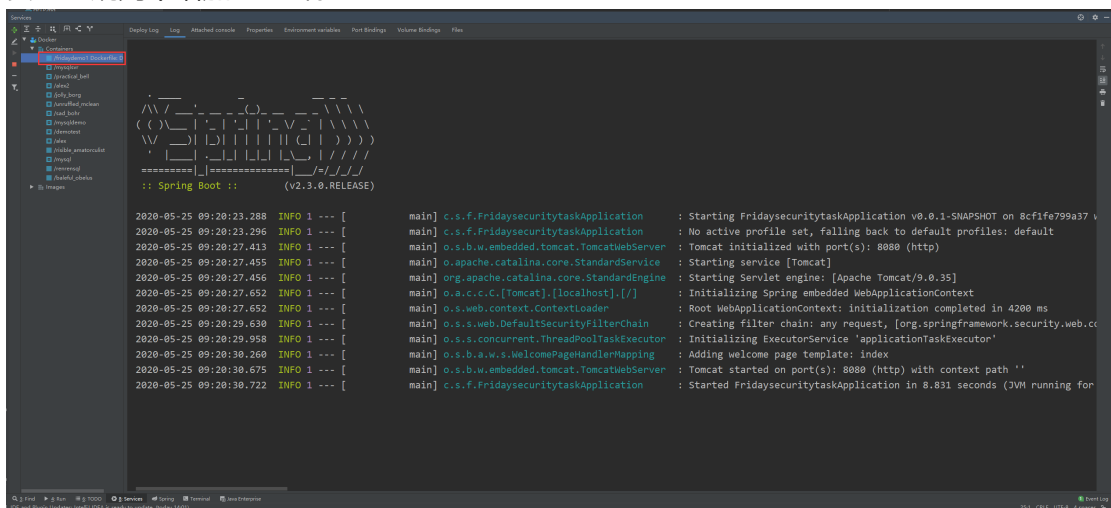
在个"Run Maven Goal"点击后，输入要执行的maven命令clean package -U -DskipTests，表示每次在构建镜像之前，都会将当前工程清理掉并且重新编译构建：

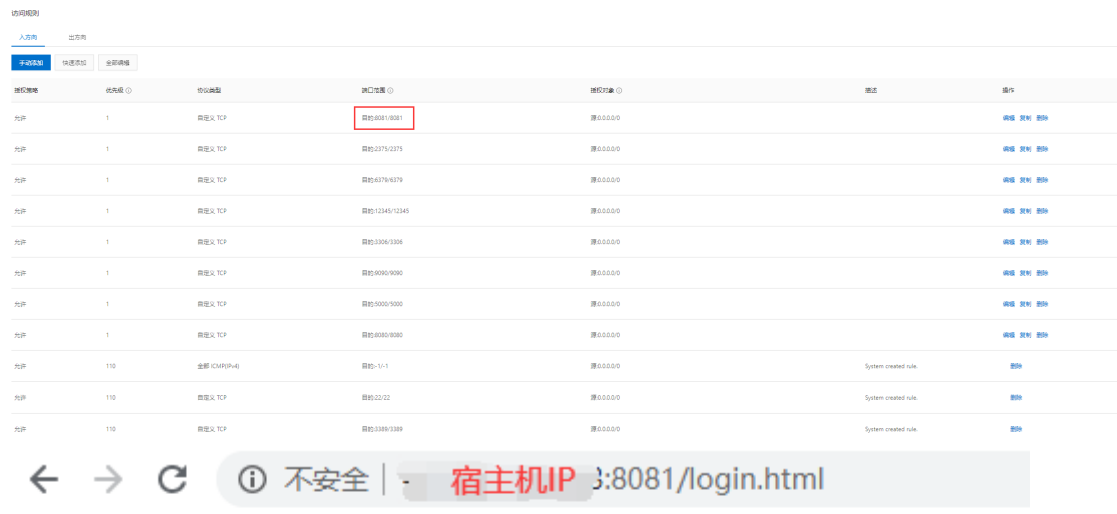


6. 点击三角按钮运行验证



启动运行成功，使用浏览器访问：<http://宿主机IP:8081>，如果是阿里云等云服务，记得在安全组规则中增加8081端口。





Friday权限管理系统

用户名称

用户密码

5.基于MySQL数据库的认证和授权

到目前为止，我们仍然只有一个可登录的用户，怎样引入多用户呢？非常简单，我们只需实现一个自定义的UserDetailsService即可。

UserDetailsService仅定义了一个loadUserByUsername方法，用于获取一个UserDetails对象。

UserDetails对象包含了一系列在验证时会用到的信息，包括用户名、密码、权限以及其他信息，Spring Security会根据这些信息判定验证是否成功。

1.创建我们的数据表，并插入三条数据，这里要注意，对密码123456我们使用的机密存储。

```
-----  
-- Table structure for users  
-----  
  
DROP TABLE IF EXISTS `users`;  
CREATE TABLE `users` (  
  `user_id` bigint NOT NULL AUTO_INCREMENT,  
  `user_name` varchar(30) COLLATE utf8mb4_general_ci NOT NULL,  
  `password` varchar(100) COLLATE utf8mb4_general_ci DEFAULT NULL,  
  `status` char(1) COLLATE utf8mb4_general_ci NOT NULL DEFAULT '0' COMMENT '0
```

```

正常1停用',
`roles` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '多个
角色用逗号间隔',
PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_general_ci;

--
-----
-- Records of users
--
-----

INSERT INTO `users` VALUES ('1', 'admin',
'$2a$10$nNQI9Ij1rU5NG9JFLQphweT0teCX60211Nysrg2V5rRSGDRmRWtm.', '0',
'ROLE_ADMIN,ROLE_USER');
INSERT INTO `users` VALUES ('2', 'user',
'$2a$10$nNQI9Ij1rU5NG9JFLQphweT0teCX60211Nysrg2V5rRSGDRmRWtm.', '0',
'ROLE_USER');
INSERT INTO `users` VALUES ('3', 'alex',
'$2a$10$nNQI9Ij1rU5NG9JFLQphweT0teCX60211Nysrg2V5rRSGDRmRWtm.', '0',
'ROLE_ADMIN,ROLE_USER');

```

2.在pom.xml中配置MySQL数据库以及Spring data jpa

```

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

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

```

3.在application.yml中配置数据库来连接参数

```

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:12345/friday?
    useUnicode=true&characterEncoding=utf8&zeroDateTimeBehavior=convertToNull&useSSL=

```



```
username: root
password: 123456
```

4.构建Users实体

```
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Users {

    /**
     * 用户ID
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column
    private Long userId;

    /**
     * 用户账号
     */
    @Column(name = "user_name")
    private String userName;

    /**
     * 密码
     */
    @Column(name = "password")
    private String password;

    /**
     * 帐号状态（0正常 1停用）
     */
    @Column(name = "status")
    private String status;

    /**
     * 用户角色（多角色用逗号间隔）
     */
    @Column(name = "roles")
    private String roles;
}
```

5.Users实体实现UserDetails接口，实现UserDetails定义的几个方法：

⊗ isAccountNonExpired、isAccountNonLocked 和 isCredentialsNonExpired 暂且用不到，统一返回

true，否则Spring Security会认为账号异常。

⊗ isEnabled对应enable字段，将其代入即可。

⊗ getAuthorities方法本身对应的是roles字段，但由于结构不一致，所以此处新建一个，并在后续进行填充。

```
/**
 * 用户对象 users
 */
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Users implements UserDetails {
    /**
     * 用户ID
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column
    private Long userId;

    /**
     * 用户账号
     */
    @Column(name = "user_name")
    private String userName;

    /**
     * 密码
     */
    @Column(name = "password")
    private String password;

    /**
     * 帐号状态（0正常 1停用）
     */
    @Column(name = "status")
    private String status;
```

```
/**
 * 用户角色（多角色用逗号间隔）
 */
@Column(name = "roles")
private String roles;

//实体类中使想要添加表中不存在字段，就要使用@Transient这个注解了。
@Transient
private List<GrantedAuthority> authorities;

public void setAuthorities(List<GrantedAuthority> authorities) {
    this.authorities = authorities;
}

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return this.authorities;
}

@Override
public String getUsername() {
    return this.userName;
}

/**
 * 账户是否未过期, 过期无法验证
 */
@Override
public boolean isAccountNonExpired() {
    return true;
}

/**
 * 指定用户是否解锁, 锁定的用户无法进行身份验证
 *
 * @return
 */
@Override
public boolean isAccountNonLocked() {
    return true;
}

/**
 * 指示是否已过期的用户的凭据(密码), 过期的凭据防止认证
 *

```

```

    * @return
    */
@Override
public boolean isCredentialsNonExpired() {
    return true;
}

/**
 * 是否可用，禁用的用户不能身份验证
 *
 * @return
 */
@Override
public boolean isEnabled() {
    return true;
}

```

6.编写接口UserDAO，继承JpaRepository<T, ID>，

```

@Repository
public interface UserDAO extends JpaRepository<Users, Long> {
}

```

7.我们需要根据输入用户名在数据库中查询User数据，然后和输入的密码作比较，现在我们来实现根据用户名查找User数据的代码，新增UserService以及它的实现类。

```

public interface UserService {
    public Users selectUserByUserName(String userName);
}

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDAO userDAO;

    @Override
    public Users selectUserByUserName(String userName) {
        Users user = new Users();
        user.setUserName(userName);
        List<Users> list = userDAO.findAll(Example.of(user));
        return list.isEmpty() ? null : list.get(0);
    }
}

```

```
}  
}
```

8.实现UserService逻辑

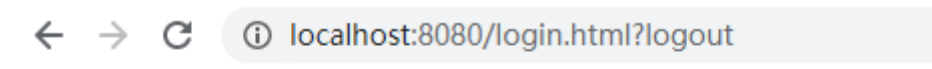
```
@Service  
public class UserDetailsServiceImpl implements UserDetailsService {  
    @Autowired  
    private UserService userService;  
  
    @Override  
    public UserDetails loadUserByUsername(String username) throws  
    UsernameNotFoundException {  
        Users users = userService.selectUserByUserName(username);  
        if (users == null) {  
            throw new UsernameNotFoundException("登录用户: " + username + "  
不存在");  
        }  
        //将数据库的roles解析为UserDetails的权限集  
        //AuthorityUtils.commaSeparatedStringToAuthorityList将逗号分隔的字符  
        集转成权限对象列表  
  
        users.setAuthorities(AuthorityUtils.commaSeparatedStringToAuthorityList(users.get  
  
        return users;  
    }  
}
```

9.修改SecurityConfig文件，将之前的内容认证方式注销掉，使用UserService逻辑来实现登录逻辑认证。

```
@Override  
public void configure(AuthenticationManagerBuilder auth) throws Exception  
{  
  
    auth.userDetailsService(myUserDetailsService).passwordEncoder(bCryptPasswordEncoder)  
  
}  
  
// @Override  
// protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {  
//     auth.inMemoryAuthentication()
```

```
//          .withUser("user")
//          .password(bCryptPasswordEncoder().encode("123456"))
//          .roles("USER")
//          .and()
//          .withUser("admin")
//          .password(bCryptPasswordEncoder().encode("123456"))
//          .roles("ADMIN")
//          .and()
//          .passwordEncoder(bCryptPasswordEncoder()); //配置BCrypt加密
//      }
```

10.启动运行

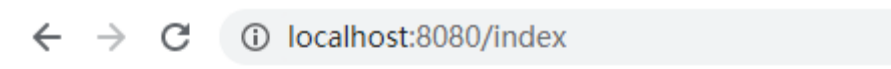


Friday权限管理系统

用户名称

用户密码

登陆成功

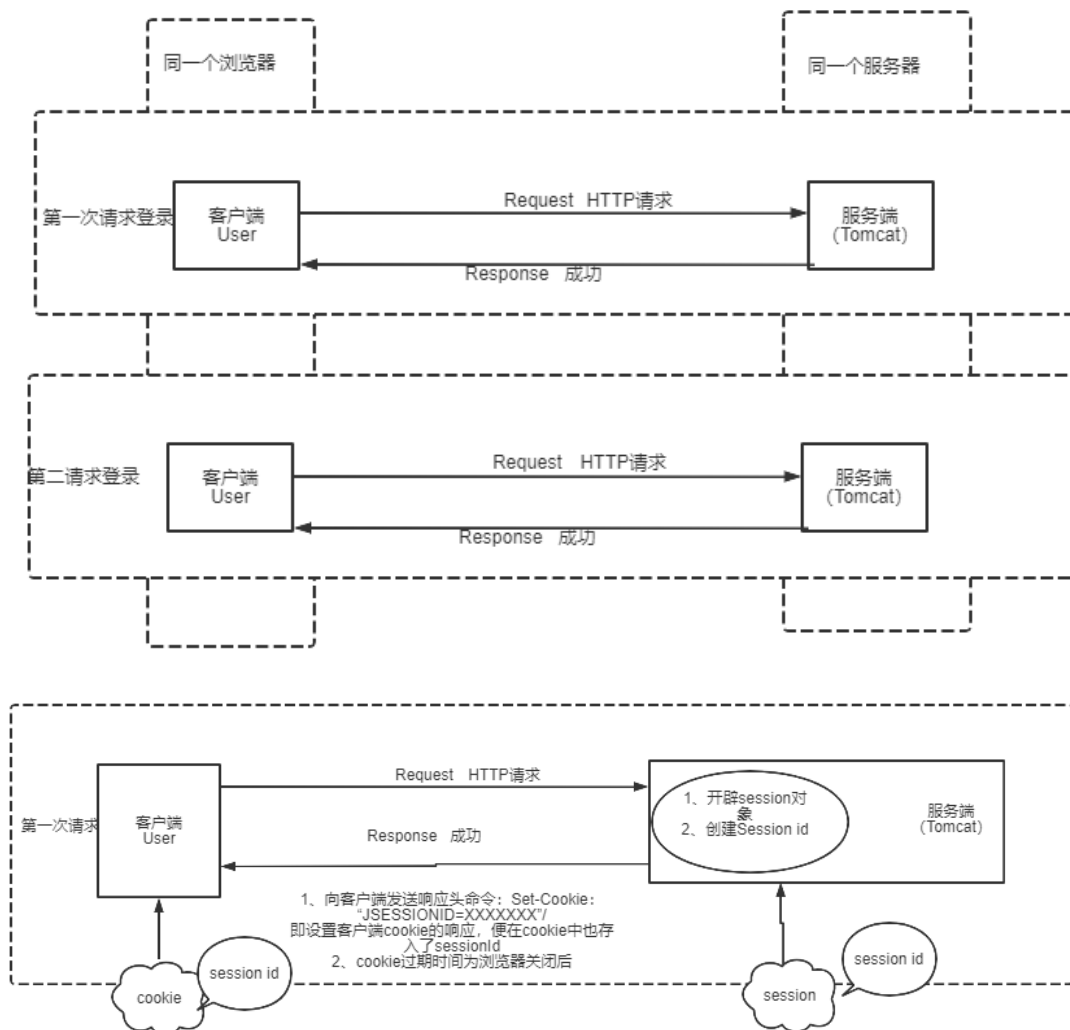


Friday权限管理系统

[退出](#)
[用户管理](#)
[角色管理](#)
[菜单管理](#)
[订单管理](#)

Spring Security提供4种方式精确的控制会话的创建:

理解会话



会话（session）就是无状态的 HTTP 实现用户状态可维持的一种解决方案。HTTP 本身的无状态使得用户在与服务器的交互过程中，每个请求之间都没有关联性。这意味着用户的访问没有身份记录，站点也无法为用户提供个性化的服务。session的诞生解决了这个难题，服务器通过与用户约定每个请求都携带一个id类的信息，从而让不同请求之间有了关联，而id又可以很方便地绑定具体用户，所以我们可以把不同请求归类到同一用户。基于这个方案，为了让用户每个请求都携带同一个id，在不妨碍体验的情况下，cookie是很好的载体。当用户首次访问系统时，系统会为该用户生成一个sessionId，并添加到cookie中。在该用户的会话期内，每个请求都自动携带该cookie，因此系统可以很轻易地识别出这是来自哪个用户的请求。

4种方式控制会话

- always: 如果当前请求没有session存在，Spring Security创建一个session；
- ifRequired（默认）：Spring Security在需要时才创建；
- sessionnever: Spring Security将永远不会主动创建session，但是如果session已经存在，它将使用该session；
- stateless: Spring Security不会创建或使用任何session。适合于接口型的无状态应用，该方式节省资源。

