Springboot配置Spring Security

最初参考:

<https://www.qikegu.com/docs/2570>

之后把代码整个重构了一遍

# 概述

本篇文章通过整合Spring security,实现了web登录功能, 具体为:

1. 向/signIn页面POST用户信息(body中携带), 服务器返回Token
2. 之后用户请求的header中带Token, 以访问需要权限的页面

# 配置

## Gradle

// https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security  
compile group: 'org.springframework.boot', name: 'spring-boot-starter-security', version: '2.2.0.RELEASE'  
//JWT  
// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt  
compile group: 'io.jsonwebtoken', name: 'jjwt', version: '0.9.1'  
// https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api  
compile group: 'javax.xml.bind', name: 'jaxb-api', version: '2.3.1'

第二个jjwt用于生成Token

第三个jaxb.api被jjwt调用, 没有不行

## yaml:

增加字段:

jwt:  
 secret: wangzilin  
 expiration: 1800

1. 用于生成Token时进行加密的密钥
2. 过期时间

# 配置Spring Security类

## 注入字段:

*/\*\*  
 \* 权限不足错误信息回调:认证错误, 鉴权错误  
 \*/*final private TokenAuthFilter tokenAuthFilter;  
  
*/\*\*  
 \* 过滤所有网络请求，从http头部Authorization字段读取token并校验  
 \*/*final private AuthErrorHandler authErrorHandler;  
*/\*\*  
 \* 用户读取用户数据  
 \*/*final private UserService userService;

## 密码加密器:

*/\*\*  
 \* 用户密码加密器  
 \* 用户的明文密码会使用该加密器进行加密, 之后与数据库进行比对  
 \*  
 \** ***@return*** *加密器  
 \*/*@Bean  
public PasswordEncoder encoder() {  
 return new BCryptPasswordEncoder();  
}

## 暴露认证管理器

*/\*\*  
 \* 获取AuthenticationManager（认证管理器），****供其他地方使用*** *\*  
 \** ***@return*** *.  
 \** ***@throws*** *Exception .  
 \*/*@Bean(name = "authenticationMangerBean")  
@Override  
public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
}

## 配置用户服务

*/\*\*  
 \* 返回一个userService, 用于导入用户数据  
 \*  
 \** ***@param*** *authenticationManagerBuilder .  
 \** ***@throws*** *Exception .  
 \*/*@Override  
public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {  
 authenticationManagerBuilder.userDetailsService(userService);  
}

这个userService是该类初始化时被注入的, userService具有通过用户名获取用户详情的功能

## 配置过滤规则/错误处理器/验证过滤器

*/\*\*  
 \* 配置过滤规则  
 \*  
 \** ***@param*** *httpSecurity .  
 \** ***@throws*** *Exception .  
 \*/*@Override  
protected void configure(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity  
 //基于token, 不需要csrf???  
 .csrf().disable()  
 // 基于token，所以不需要session  
 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*)  
 .and()  
 // 设置myUnauthorizedHandler处理认证失败、鉴权失败  
 .exceptionHandling().authenticationEntryPoint(authErrorHandler).accessDeniedHandler(authErrorHandler)  
 .and()  
 //下面开始设置权限  
 .authorizeRequests()  
 .antMatchers("/hello/\*\*").authenticated()  
 .antMatchers("/chat/\*\*").authenticated()  
 .antMatchers("/card/\*\*").authenticated()  
 .anyRequest().permitAll();  
  
 httpSecurity.addFilterBefore(tokenAuthFilter, UsernamePasswordAuthenticationFilter.class);  
 // 禁用缓存  
 httpSecurity.headers().cacheControl();  
  
}

## 跨源访问

*/\*\*  
 \* 配置跨源访问  
 \*  
 \** ***@return*** *.  
 \*/*@Bean  
CorsConfigurationSource corsConfigurationSource() {  
 UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", new CorsConfiguration().applyPermitDefaultValues());  
 return source;  
}

## 合并代码:

@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true)//这样就可以在Controller上配置权限  
public class Security extends WebSecurityConfigurerAdapter {  
  
 */\*\*  
 \* 权限不足错误信息处理:认证错误, 鉴权错误  
 \*/* final private AuthErrorHandler authErrorHandler;  
  
 */\*\*  
 \* 过滤器, jwt校验过滤器，从http头部Authorization字段读取token并校验  
 \*/* final private JwtAuthFilter jwtAuthFilter;  
 */\*\*  
 \* 用户读取用户数据  
 \*/* final private UserService userService;  
  
 public Security(AuthErrorHandler authErrorHandler, JwtAuthFilter jwtAuthFilter, UserService userService) {  
 this.authErrorHandler = authErrorHandler;  
 this.jwtAuthFilter = jwtAuthFilter;  
 this.userService = userService;  
 }  
  
  
 */\*\*  
 \* 用户密码加密器  
 \* 用户的明文密码会使用该加密器进行加密, 之后与数据库进行比对  
 \*  
 \** ***@return*** *加密器  
 \*/* @Bean  
 public PasswordEncoder encoder() {  
 return new BCryptPasswordEncoder();  
 }  
  
  
 */\*\*  
 \* 获取AuthenticationManager（认证管理器），可以在其他地方使用  
 \*  
 \** ***@return*** *.  
 \** ***@throws*** *Exception .  
 \*/* @Bean(name = "authenticationMangerBean")  
 @Override  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
  
 */\*\*  
 \* 返回一个userService, 用于导入用户数据  
 \*  
 \** ***@param*** *authenticationManagerBuilder .  
 \** ***@throws*** *Exception .  
 \*/* @Override  
 public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {  
 authenticationManagerBuilder.userDetailsService(userService);  
 }  
  
  
 */\*\*  
 \* 配置过滤规则  
 \*  
 \** ***@param*** *httpSecurity .  
 \** ***@throws*** *Exception .  
 \*/* @Override  
 protected void configure(HttpSecurity httpSecurity) throws Exception {  
 httpSecurity  
 //基于token, 不需要csrf???  
 .csrf().disable()  
 // 基于token，所以不需要session  
 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*)  
 .and()  
 // 设置myUnauthorizedHandler处理认证失败、鉴权失败  
 .exceptionHandling().authenticationEntryPoint(authErrorHandler).accessDeniedHandler(authErrorHandler)  
 .and()  
 //下面开始设置权限  
 .authorizeRequests()  
 .antMatchers("/hello/\*\*").authenticated()  
 .antMatchers("/chat/\*\*").authenticated()  
 .antMatchers("/card/\*\*").authenticated()  
 .anyRequest().permitAll();  
  
 httpSecurity.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);  
 // 禁用缓存  
 httpSecurity.headers().cacheControl();  
  
 }  
  
  
 */\*\*  
 \* 配置跨源访问  
 \*  
 \** ***@return*** *.  
 \*/* @Bean  
 CorsConfigurationSource corsConfigurationSource() {  
 UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
 source.registerCorsConfiguration("/\*\*", new CorsConfiguration().applyPermitDefaultValues());  
 return source;  
 }  
}

# 创建验证错误处理器

验证错误处理器在请求验证错误时被回调.

本来有两种认证错误, 认证失败和鉴权失败. 为了代码简单, 把这两种失败的回调函数放到一个类里面.

只要实现这个类的时候implement两个接口即可.

/\*\*  
 \* 当认证失败，系统会抛出认证失败异常，可以配置我们自己的认证失败处理类，同样鉴权失败也可以配置我们自己的失败处理类。  
 \* JwtAuthError继承AuthenticationEntryPoint（认证失败接口）、AccessDeniedHandler（鉴权失败接口），  
 \* 重写了这2个接口类的失败处理方法，其实JwtAuthError可以分为2个类，我们合二为一了  
 \*/  
@Component  
public class AuthErrorHandler implements AuthenticationEntryPoint, AccessDeniedHandler {  
  
 private static final org.slf4j.Logger log = LoggerFactory.getLogger(AuthErrorHandler.class);  
  
 /\*\*  
 \* 认证失败处理，返回401 json数据  
 \*  
 \* **@param** request .  
 \* **@param** response .  
 \* **@param** authException .  
 \* **@throws** IOException .  
 \*/  
 @Override  
 public void commence(HttpServletRequest request, HttpServletResponse response,  
 AuthenticationException authException) throws IOException {  
 log.info("认证失败处理，返回401");  
 response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);  
 response.setContentType("application/json;charset=utf-8");  
 response.getWriter().write("{\"status\":401,\"message\":\"Unauthorized or invalid token\"}");  
 }  
  
 /\*\*  
 \* 鉴权失败处理，返回403 json数据  
 \*  
 \* **@param** request .  
 \* **@param** response .  
 \* **@param** accessDeniedException .  
 \* **@throws** IOException .  
 \*/  
 @Override  
 public void handle(HttpServletRequest request, HttpServletResponse response,  
 AccessDeniedException accessDeniedException) throws IOException {  
 log.info("鉴权失败处理，返回403");  
 response.setStatus(HttpServletResponse.SC\_FORBIDDEN);  
 response.setContentType("application/json;charset=utf-8");  
 response.getWriter().write("{\"status\":403,\"message\":\"Forbidden\"}");  
 }  
}

# 创建用户验证POJO类

spring security使用一个特定的用户POJO类来储存用户数据:

这个POJO类必须实现UserDetails接口:

以下类的构造方法为了简便, 写死了一些字段:

public class UserForAuth implements UserDetails {  
  
 private String userId;  
 private String password;  
 private boolean accountExpired;  
 private boolean accountLocked;  
 private boolean credentialExpired;  
 private boolean enabled;  
 private Set<GrantedAuthority> authorities;  
  
  
 //把从数据库读取的用户对象转换为为用户验证的对象.  
 public UserForAuth(UserProfile userProfile) {  
 this(userProfile.getUserId(), userProfile.getPassword());  
 }  
  
 public UserForAuth(String userId, String password) {  
  
 //这里将所有用户的全设为USER  
 Set<GrantedAuthority> authorities = new HashSet<>();  
 authorities.add(new SimpleGrantedAuthority("ROLE\_USER"));//ROLE前缀必须  
  
 this.userId = userId;  
 this.password = password;  
  
 this.authorities = authorities;  
  
 this.accountExpired = false;  
 this.accountLocked = false;  
 this.credentialExpired = false;  
 this.enabled = true;  
 }  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 return authorities;  
 }  
  
 @Override  
 public String getPassword() {  
 return password;  
 }  
  
 @Override  
 public String getUsername() {  
 return userId;  
 }  
  
 @Override  
 public boolean isAccountNonExpired() {  
 return !accountExpired;  
 }  
  
 @Override  
 public boolean isAccountNonLocked() {  
 return !accountLocked;  
 }  
  
 @Override  
 public boolean isCredentialsNonExpired() {  
 return !credentialExpired;  
 }  
  
 @Override  
 public boolean isEnabled() {  
 return enabled;  
 }  
  
 public String getAuthoritiesString() {  
 List<String> authorityList = new ArrayList<>();  
 authorities.forEach((authority) -> authorityList.add(authority.getAuthority()));  
 return String.*join*(",", authorityList);  
 }  
}

# 创建Token生成器

jwt用于生成token/从token中获取字段

以下代码实现了一个token的工具类

@Service  
public class JwtUtil {  
 private static final Logger *log* = LoggerFactory.*getLogger*(JwtUtil.class);  
  
 private io.jsonwebtoken.Clock clock = DefaultClock.*INSTANCE*;  
  
 @Value("${jwt.secret}")  
 private String secret;  
  
 @Value("${jwt.expiration}")  
 private Long expiration;  
  
 public String getUsernameFromToken(String token) {  
 return getClaimFromToken(token, Claims::getSubject);  
 }  
  
 public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {  
 final Claims claims = getAllClaimsFromToken(token);  
 return claimsResolver.apply(claims);  
 }  
  
 public <T> T getClaimFromToken(String token, String claimName, Class<T> requiredType) {  
 final Claims claims = getAllClaimsFromToken(token);  
 return claims.get(claimName, requiredType);  
 }  
  
 private Claims getAllClaimsFromToken(String token) {  
 return Jwts.*parser*()  
 .setSigningKey(secret)  
 .parseClaimsJws(token)  
 .getBody();  
 }  
  
 private Boolean isTokenExpired(String token) {  
 final Date expiration = getExpirationDateFromToken(token);  
 return expiration.before(clock.now());  
 }  
  
 private Boolean isCreatedBeforeLastPasswordReset(Date created, Date lastPasswordReset) {  
 return (lastPasswordReset != null && created.before(lastPasswordReset));  
 }  
  
 private Boolean ignoreTokenExpiration(String token) {  
 // here you specify tokens, for that the expiration is ignored  
 return false;  
 }  
  
 */\*\*  
 \* 生成Token Token中放入:  
 \* - 用户名  
 \* - 创建时间  
 \* - 过期时间  
 \*  
 \** ***@param*** *user 验证用户实体  
 \** ***@return*** *token  
 \*/* public String generateToken(UserForAuth user) {  
 final Date createdDate = clock.now();  
 final Date expirationDate = calculateExpirationDate(createdDate);  
  
 return Jwts.*builder*()  
 .setSubject(user.getUsername())  
 .setIssuedAt(createdDate)  
 .setExpiration(expirationDate)  
 .signWith(SignatureAlgorithm.*HS512*, secret)  
 .compact();  
 }  
  
  
 public Boolean canTokenBeRefreshed(String token, Date lastPasswordReset) {  
 final Date created = getIssuedAtDateFromToken(token);  
 return !isCreatedBeforeLastPasswordReset(created, lastPasswordReset)  
 && (!isTokenExpired(token) || ignoreTokenExpiration(token));  
 }  
  
 public String refreshToken(String token) {  
 final Date createdDate = clock.now();  
 final Date expirationDate = calculateExpirationDate(createdDate);  
  
 final Claims claims = getAllClaimsFromToken(token);  
 claims.setIssuedAt(createdDate);  
 claims.setExpiration(expirationDate);  
  
 return Jwts.*builder*()  
 .setClaims(claims)  
 .signWith(SignatureAlgorithm.*HS512*, secret)  
 .compact();  
 }  
  
  
 */\*\*  
 \* 计算token过期时间  
 \*  
 \** ***@param*** *createdDate 创建时间  
 \** ***@return*** *过期时间  
 \*/* private Date calculateExpirationDate(Date createdDate) {  
 return new Date(createdDate.getTime() + expiration \* 1000);  
 }  
  
}

以上代码中:

1. 使用generateToken生成Token(用户名是方法参数)
2. 使用getUsernameFromToken从Token中获取用户名

# 配置用户服务

为了能够被springSecurity使用, 用户服务必须实现UserDetailsService接口:

*/\*\*  
 \* 这个类负责处理AuthController,接收验证用户的请求  
 \*/*@Service  
public class UserService implements UserDetailsService {  
  
 private UserDAO userDAO;  
  
 private JwtUtil jwtUtil;  
  
 final private static org.slf4j.Logger *log* = LoggerFactory.*getLogger*(UserService.class);  
  
 public UserService(UserDAO userDAO, JwtUtil jwtUtil) {  
 this.userDAO = userDAO;  
 this.jwtUtil = jwtUtil;  
 }  
  
  
 public void addUser(SignRequest signRequest) {  
 this.addUser(new UserProfile(signRequest.getUserId(), signRequest.getPassword()));  
 }  
  
  
 public void addUser(UserProfile userProfile) {  
 userDAO.addUser(userProfile);  
 }  
  
  
 @Override  
 public UserForAuth loadUserByUsername(String username) throws UsernameNotFoundException {  
 UserProfile userProfile = userDAO.findUser(username);  
 //将从数据库获取的user转为专为用户验证的user  
 return new UserForAuth(userProfile);  
 }  
  
 public String signIn(String userId, String password) {  
 // 认证用户，认证失败抛出异常，由JwtAuthError的commence类返回401  
 UsernamePasswordAuthenticationToken upToken = new UsernamePasswordAuthenticationToken(userId, password);  
  
 final Authentication authentication =  
 ((AuthenticationManager) BeanUtil.*getBean*("authenticationMangerBean")).authenticate(upToken);  
 //这步在内部调用了UserService来验证用户是否存在  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
  
 // 如果认证通过，再次访问数据库, 取出用户  
 final UserForAuth userForAuth = loadUserByUsername(userId);  
 return jwtUtil.generateToken(userForAuth);  
 }  
  
}

这个类一定要实现loadUserByUsername方法, 这个方法根据传入的用户名返回上一节定义的用户验证POJO类

值得一提的是, 本类类还可以承接controller的需求, 实现用户的登录,即signIn方法:

public String signIn(String userId, String password) {  
 // 认证用户，认证失败抛出异常，由JwtAuthError的commence类返回401  
 UsernamePasswordAuthenticationToken upToken = new UsernamePasswordAuthenticationToken(userId, password);  
  
 final Authentication authentication =  
 ((AuthenticationManager) BeanUtil.*getBean*("authenticationMangerBean")).authenticate(upToken);  
 //这步在内部调用了UserService来验证用户是否存在  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
  
 // 如果认证通过，再次访问数据库, 取出用户  
 final UserForAuth userForAuth = loadUserByUsername(userId);  
 return jwtUtil.generateToken(userForAuth);  
}

在这个方法中注意这一行:

((AuthenticationManager) BeanUtil.*getBean*("authenticationMangerBean")).authenticate(upToken)

它使用一个工具类调用了在3.3中暴露的验证管理器.这个验证管理器在**内部**调用了本类的loadUserByUsername方法

注意:在本类中不能直接注入AuthenticationManager, 这样会导致循环注入问题,因为AuthenticationManager内部又会注入本类. 只能使用工具类来调用该Bean.

在这个类的最后, 使用第5章创建的Token生成器根据用户名生成了Token

# 配置验证过滤器

在3.5中配置了验证过滤器:

httpSecurity.addFilterBefore(tokenAuthFilter, UsernamePasswordAuthenticationFilter.class);

接下来是该验证过滤器的实现:

@Component  
public class TokenAuthFilter extends OncePerRequestFilter {  
  
 private static final org.slf4j.Logger *log* = LoggerFactory.*getLogger*(TokenAuthFilter.class);  
  
 final private JwtUtil jwtUtil;  
  
 final private UserService userService;  
  
 final private String tokenHeader = "Authorization";  
  
 public TokenAuthFilter(JwtUtil jwtUtil, UserService userService) {  
 this.jwtUtil = jwtUtil;  
 this.userService = userService;  
 }  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,  
 FilterChain filterChain) throws ServletException, IOException {  
 // 从http头部读取jwt  
 String authToken = request.getHeader(this.tokenHeader);  
 if (authToken != null) {  
 UserForAuth userForAuth = null;  
 // 从jwt中解出账号与角色信息  
 try {  
 //这里没有验证Token是否过期  
 String username = jwtUtil.getUsernameFromToken(authToken);  
 userForAuth = userService.loadUserByUsername(username);  
 } catch (Exception e) {  
 *log*.debug("异常详情", e);  
 *log*.info("无效token");  
 }  
  
 // 如果jwt正确解出账号信息，说明是合法用户，设置认证信息，认证通过  
 if (userForAuth != null && SecurityContextHolder.*getContext*().getAuthentication() == null) {  
  
 UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(  
 userForAuth.getUsername(), null, userForAuth.getAuthorities());  
  
 // 把请求的信息设置到UsernamePasswordAuthenticationToken details对象里面，包括发请求的ip等  
 auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
  
 // 设置认证信息  
 SecurityContextHolder.*getContext*().setAuthentication(auth);  
 }  
 }  
  
 // 调用下一个过滤器  
 filterChain.doFilter(request, response);  
 }  
}

注意:在第一行本类实现了OncePerRequestFilter, 这就意味着这个类在每次请求时仅被调用一次.

这个类首先检查请求的header中是否存在Token,如果存在并验证通过,则使用Securtiy的

SecurityContextHolder.getContext().setAuthentication(auth);

方法使该请求通过认证, 此时请求就可以被Controller处理了.

如果没有通过认证, 则会调用第4章实现的错误处理器方法.

# 在任意地方使用登录用户数据

归功于springboot的注入机制, 用户信息其实在任何地方都能使用, 使用方法为:

User user = (User) SecurityContextHolder.*getContext*().getAuthentication().getDetails();

那这个Detail是什么呢?

在上一章配置验证过滤器步骤中, 在解出账号信息后,有一步是:

// 把请求的信息设置到UsernamePasswordAuthenticationToken details对象里面，包括发请求的ip等  
 auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

这个setDetails里面就可以放之后用到的用户信息:

为了获得如上所述的User数据, 可以把setDetail设置为User:

// 把请求的信息设置到UsernamePasswordAuthenticationToken details对象里面，包括发请求的ip等  
// auth.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
 auth.setDetails(user);

注释原来的, setDetail中传入一个User对象即可