Spring Security做JWT认证和授权

空挡 关注
13 2018.09.22 11:39:41 字数 3,550 阅读 129,421

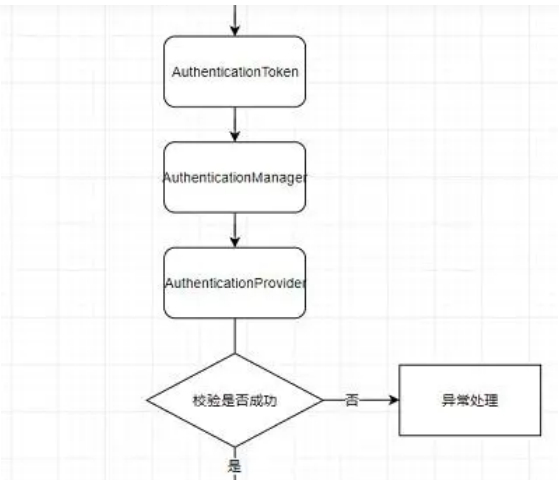
上一篇博客讲了如何使用Shiro和JWT做认证和授权（传送门：https://www.jianshu.com/p/0b1131be7ace），总的来说shiro是一个比较早期和简单的框架，这个从最近已经基本不做版本更新就可以看出来。这篇文章我们讲一下如何使用更加流行和完整的spring security来实现同样的需求。

Spring Security的架构

按照惯例，在使用之前我们先讲一下简单的架构。不知道是因为spring-security后出来还是因为优秀的设计殊途同归，对于核心模块，spring-security和shiro有80%以上的设计相似度。所以下面介绍中会多跟shiro做对比，如果你对shiro不了解也没关系，跟shiro对比的部分跳过就好。

spring-security中核心概念

- **AuthenticationManager**, 用户认证的管理类，所有的认证请求（比如login）都会通过提交一个token给 `AuthenticationManager` 的 `authenticate()` 方法来实现。当然事情肯定不是它来做，具体校验动作会由 `AuthenticationManager` 将请求转发给具体的实现类来做。根据实现反馈的结果再调用具体的Handler来给用户以反馈。这个类基本等同于shiro的 `SecurityManager` 。
- **AuthenticationProvider**, 认证的具体实现类，一个provider是一种认证方式的实现，比如提交的用户名密码我是通过和DB中查出的user记录做比对实现的，那就有一个 `DaoProvider` ；如果我是通过CAS请求单点登录系统实现，那就有一个 `CASProvider` 。这个是不是和shiro的Realm的定义很像？基本上你可以帮他们当成同一个东西。按照Spring一贯的作风，主流的认证方式它都已经提供了默认实现，比如DAO、LDAP、CAS、OAuth2等。
  前面讲了 `AuthenticationManager` 只是一个代理接口，真正的认证就是由 `AuthenticationProvider` 来做的。一个 `AuthenticationManager` 可以包含多个Provider，每个provider通过实现一个support方法来表示自己支持那种Token的认证。
  `AuthenticationManager` 默认的实现类是 `ProviderManager` 。
- **UserDetailService**, 用户认证通过Provider来做，所以Provider需要拿到系统已经保存的认证信息，获取用户信息的接口spring-security抽象成 `UserDetailService` 。虽然叫Service,但是我更愿意把它认为是我们系统里经常有的 `UserDao` 。
- **AuthenticationToken**, 所有提交给 `AuthenticationManager` 的认证请求都会被封装成一个Token的实现，比如最容易理解的 `UsernamePasswordAuthenticationToken` 。这个就不多讲了，连名字都跟Shiro中一样。
- **SecurityContext**，当用户通过认证之后，就会为这个用户生成一个唯一的 `SecurityContext` ，里面包含用户的认证信息 `Authentication` 。通过SecurityContext我们可以获取到用户的标识 `Principle` 和授权信息 `GrantedAuthrity` 。在系统的任何地方只要通过 `SecurityHolder.getSecruityContext()` 就可以获取到 `SecurityContext` 。在Shiro中通过 `SecurityUtils.getSubject()` 到达同样的目的。
  我们大概通过一个认证流程来认识下上面几个关键的概念



认证流程

对web系统的支持

毫无疑问，对于spring框架使用最多的还是web系统。对于web系统来说进入认证的最佳入口就是Filter了。spring security不仅实现了认证的逻辑，还通过filter实现了常见的web攻击的防护。

常用Filter

下面按照request进入的顺序列举一下常用的Filter：

- SecurityContextPersistenceFilter，用于将 `SecurityContext` 放入Session的Filter
- UsernamePasswordAuthenticationFilter, 登录认证的Filter,类似的还有CasAuthenticationFilter，BasicAuthenticationFilter等等。在这些Filter中生成用于认证的token，提交到AuthenticationManager，如果认证失败会直接返回。
- RememberMeAuthenticationFilter，通过cookie来实现remember me功能的Filter
- AnonymousAuthenticationFilter，如果一个请求在到达这个filter之前SecurityContext没有初始化，则这个filter会默认生成一个匿名SecurityContext。这在支持匿名用户的系统中非常有用。
- ExceptionTranslationFilter，捕获所有Spring Security抛出的异常，并决定处理方式
- FilterSecurityInterceptor， 权限校验的拦截器，访问的url权限不足时会抛出异常

  Filter的顺序

  既然用了上面那么多filter，它们在FilterChain中的先后顺序就显得非常重要了。对于每一个系统或者用户自定义的filter，spring security都要求必须指定一个order，用来做排序。对于系统的filter的默认顺序，是在一个 `FilterComparator` 类中定义的，核心实现如下。

```
    FilterComparator() {
        int order = 100;
        put(ChannelProcessingFilter.class, order);
        order += STEP;
        put(ConcurrentSessionFilter.class, order);
        order += STEP;
        put(WebAsyncManagerIntegrationFilter.class, order);
        order += STEP;
        put(SecurityContextPersistenceFilter.class, order);
        order += STEP;
        put(HeaderWriterFilter.class, order);
        order += STEP;
        put(CorsFilter.class, order);
        order += STEP;
        put(CsrfFilter.class, order);
        order += STEP;
```

```
        put(X509AuthenticationFilter.class, order);
        order += STEP;
        put(AbstractPreAuthenticatedProcessingFilter.class, order);
        order += STEP;
        filterToOrder.put("org.springframework.security.cas.web.CasAuthenticationFilte
                order);
        order += STEP;
        filterToOrder.put(
            "org.springframework.security.oauth2.client.web.OAuth2LoginAuthenticationF
            order);
        order += STEP;
        put(UsernamePasswordAuthenticationFilter.class, order);
        order += STEP;
        put(ConcurrentSessionFilter.class, order);
        order += STEP;
        filterToOrder.put(
                "org.springframework.security.openid.OpenIDAuthenticationFilter", orde
        order += STEP;
        put(DefaultLoginPageGeneratingFilter.class, order);
        order += STEP;
        put(ConcurrentSessionFilter.class, order);
        order += STEP;
        put(DigestAuthenticationFilter.class, order);
        order += STEP;
        put(BasicAuthenticationFilter.class, order);
        order += STEP;
        put(RequestCacheAwareFilter.class, order);
        order += STEP;
        put(SecurityContextHolderAwareRequestFilter.class, order);
        order += STEP;
        put(JaasApiIntegrationFilter.class, order);
        order += STEP;
        put(RememberMeAuthenticationFilter.class, order);
        order += STEP;
        put(AnonymousAuthenticationFilter.class, order);
        order += STEP;
        put(SessionManagementFilter.class, order);
        order += STEP;
        put(ExceptionTranslationFilter.class, order);
        order += STEP;
        put(FilterSecurityInterceptor.class, order);
        order += STEP;
        put(SwitchUserFilter.class, order);
    }
```

对于用户自定义的filter，如果要加入spring security 的FilterChain中，必须指定加到已有的那个filter之前或者之后，具体下面我们用到自定义filter的时候会说明。

JWT认证的实现

关于使用JWT认证的原因，上一篇介绍Shiro的文章中已经说过了，这里不再多说。需求也还是那3个：

- 支持用户通过用户名和密码登录
- 登录后通过http header返回token，每次请求，客户端需通过header将token带回，用于权限校验
- 服务端负责token的定期刷新

  下面我们直接进入Spring Secuiry的项目搭建。

项目搭建

gradle配置

最新的spring项目开始默认使用gradle来做依赖管理了，所以这个项目也尝试下gradle的配置。除了springmvc和security的starter之外，还依赖了auth0的jwt工具包。JSON处理使用了fastjson。

```
        springBootVersion = '2.0.4.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVers
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group = 'com.github.springboot'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}


dependencies {
    compile('org.springframework.boot:spring-boot-starter-security')
    compile('org.springframework.boot:spring-boot-starter-web')
    compile('org.apache.commons:commons-lang3:3.8')
    compile('com.auth0:java-jwt:3.4.0')
    compile('com.alibaba:fastjson:1.2.47')

    testCompile('org.springframework.boot:spring-boot-starter-test')
    testCompile('org.springframework.security:spring-security-test')
}
```

登录认证流程

Filter

对于用户登录行为，security通过定义一个Filter来拦截/login来实现的。spring security默认支持form方式登录，所以对于使用json发送登录信息的情况，我们自己定义一个Filter，这个Filter直接从 `AbstractAuthenticationProcessingFilter` 继承，只需要实现两部分，一个是RequestMatcher，指名拦截的Request类型；另外就是从json body中提取出username和password提交给AuthenticationManager。

```
public class MyUsernamePasswordAuthenticationFilter extends AbstractAuthenticationProc

    public MyUsernamePasswordAuthenticationFilter() {
         //拦截url为 "/login" 的POST请求
        super(new AntPathRequestMatcher("/login", "POST"));
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServle
            throws AuthenticationException, IOException, ServletException {
        //从json中获取username和password
        String body = StreamUtils.copyToString(request.getInputStream(), Charset.forNa
        String username = null, password = null;
        if(StringUtils.hasText(body)) {
            JSONObject jsonObj = JSON.parseObject(body);
            username = jsonObj.getString("username");
            password = jsonObj.getString("password");
        }

        if (username == null)
            username = "";
        if (password == null)
            password = "";
        username = username.trim();
       //封装到token中提交
        UsernamePasswordAuthenticationToken authRequest = new UsernamePasswordAuthenti
                username, password);

```

Provider

前面的流程图中讲到了，封装后的token最终是交给provider来处理的。对于登录的provider，spring security已经提供了一个默认实现 `DaoAuthenticationProvider` 我们可以直接使用，这个类继承了 `AbstractUserDetailsAuthenticationProvider` 我们来看下关键部分的源代码是怎么做的。

```
public abstract class AbstractUserDetailsAuthenticationProvider implements
        AuthenticationProvider, InitializingBean, MessageSourceAware {
...
    //这个方法返回true，说明支持该类型的token
    public boolean supports(Class<?> authentication) {
        return (UsernamePasswordAuthenticationToken.class
                .isAssignableFrom(authentication));
   }
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
             ...

            try {
            // 获取系统中存储的用户信息
                user = retrieveUser(username,
                        (UsernamePasswordAuthenticationToken) authentication);
            }
            catch (UsernameNotFoundException notFound) {
                logger.debug("User '" + username + "' not found");

                if (hideUserNotFoundExceptions) {
                    throw new BadCredentialsException(messages.getMessage(
                            "AbstractUserDetailsAuthenticationProvider.badCredentials"
                            "Bad credentials"));
                }
                else {
                    throw notFound;
                }
            }


        }

        try {
           //检查user是否已过期或者已锁定
            preAuthenticationChecks.check(user);
           //将获取到的用户信息和登录信息做比对
            additionalAuthenticationChecks(user,
                    (UsernamePasswordAuthenticationToken) authentication);
        }
        catch (AuthenticationException exception) {
            ...
            throw exception;
        }
        ...
        //如果认证通过，则封装一个AuthenticationInfo，放到SecurityContext中
        return createSuccessAuthentication(principalToReturn, authentication, user);
    }

...
}
```

上面的代码中，核心流程就是 `retrieveUser()` 获取系统中存储的用户信息，再对用户信息做了过期和锁定等校验后交给 `additionalAuthenticationChecks()` 和用户提交的信息做比对。
这两个方法我们看他的继承类 `DaoAuthenticationProvider` 是怎么实现的。

```
public class DaoAuthenticationProvider extends AbstractUserDetailsAuthenticationProvid
    /**
     * 加密密码比对
     */
     protected void additionalAuthenticationChecks(UserDetails userDetails,
            UsernamePasswordAuthenticationToken authentication)
            throws AuthenticationException {
        if (authentication.getCredentials() == null) {
            logger.debug("Authentication failed: no credentials provided");
```

```
        String presentedPassword = authentication.getCredentials().toString();

        if (!passwordEncoder.matches(presentedPassword, userDetails.getPassword())) {
            logger.debug("Authentication failed: password does not match stored value"

            throw new BadCredentialsException(messages.getMessage(
                    "AbstractUserDetailsAuthenticationProvider.badCredentials",
                    "Bad credentials"));
        }
    }
    /**
     * 系统用户获取
     */
    protected final UserDetails retrieveUser(String username,
            UsernamePasswordAuthenticationToken authentication)
            throws AuthenticationException {
        prepareTimingAttackProtection();
        try {
            UserDetails loadedUser = this.getUserDetailsService().loadUserByUsername(u
            if (loadedUser == null) {
                throw new InternalAuthenticationServiceException(
                        "UserDetailsService returned null, which is an interface contr
            }
            return loadedUser;
        }
        catch (UsernameNotFoundException ex) {
            mitigateAgainstTimingAttack(authentication);
            throw ex;
        }
        catch (InternalAuthenticationServiceException ex) {
            throw ex;
        }
        catch (Exception ex) {
            throw new InternalAuthenticationServiceException(ex.getMessage(), ex);
        }
    }
}
```

上面的方法实现中，用户获取是调用了 `UserDetailsService` 来完成的。这个是一个只有一个方法的接口，所以我们自己要做的，就是将自己的 `UserDetailsService` 实现类配置成一个Bean。下面是实例代码，真正的实现需要从数据库或者缓存中获取。

```
public class JwtUserService implements UserDetailsService{
    //真实系统需要从数据库或缓存中获取，这里对密码做了加密
     return User.builder().username("Jack").password(passwordEncoder.encode("jack-pass
}
```

我们再来看另外一个密码比对的方法，也是委托给一个 `PasswordEncoder` 类来实现的。一般来说，存在数据库中的密码都是要经过加密处理的，这样万一数据库数据被拖走，也不会泄露密码。spring一如既往的提供了主流的加密方式，如MD5,SHA等。如果不显示指定的话，Spring会默认使用 `BCryptPasswordEncoder` ，这个是目前相对比较安全的加密方式。具体介绍可参考spring-security 的官方文档 - Password Endcoding

认证结果处理

filter将token交给provider做校验，校验的结果无非两种，成功或者失败。对于这两种结果，我们只需要实现两个Handler接口，set到Filter里面，Filter在收到Provider的处理结果后会回调这两个Handler的方法。

先来看成功的情况，针对jwt认证的业务场景，登录成功需要返回给客户端一个token。所以成功的handler的实现类中需要包含这个逻辑。

```
public class JsonLoginSuccessHandler implements AuthenticationSuccessHandler{

    private JwtUserService jwtUserService;

    public JsonLoginSuccessHandler(JwtUserService jwtUserService) {
        this.jwtUserService = jwtUserService;
    }
```

```
        response.setHeader("Authorization", token);
    }

}
```

再来看失败的情况，登录失败比较简单，只需要回复一个401的Response即可。

```
public class HttpStatusLoginFailureHandler implements AuthenticationFailureHandler{
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletRespons
            AuthenticationException exception) throws IOException, ServletException {
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
    }
}
```

JsonLoginConfigurer

以上整个登录的流程的组件就完整了，我们只需要把它们组合到一起就可以了。这里继承一个 `AbstractHttpConfigurer` ，对Filter做配置。

```
public class JsonLoginConfigurer<T extends JsonLoginConfigurer<T, B>, B extends HttpSe

    private MyUsernamePasswordAuthenticationFilter authFilter;

    public JsonLoginConfigurer() {
        this.authFilter = new MyUsernamePasswordAuthenticationFilter();
    }

    @Override
    public void configure(B http) throws Exception {
        //设置Filter使用的AuthenticationManager,这里取公共的即可
        authFilter.setAuthenticationManager(http.getSharedObject(AuthenticationManager
        //设置失败的Handler
        authFilter.setAuthenticationFailureHandler(new HttpStatusLoginFailureHandler()
        //不将认证后的context放入session
        authFilter.setSessionAuthenticationStrategy(new NullAuthenticatedSessionStrate

        MyUsernamePasswordAuthenticationFilter filter = postProcess(authFilter);
        //指定Filter的位置
        http.addFilterAfter(filter, LogoutFilter.class);
    }
    //设置成功的Handler，这个handler定义成Bean，所以从外面set进来
    public JsonLoginConfigurer<T,B> loginSuccessHandler(AuthenticationSuccessHandler a
        authFilter.setAuthenticationSuccessHandler(authSuccessHandler);
        return this;
    }

}
```

这样Filter就完整的配置好了，当调用configure方法时，这个filter就会加入security FilterChain的指定位置。这个是在全局定义的地方，我们放在最后说。在全局配置的地方，也会将 `DaoAuthenticationProvider` 放到 `ProviderManager` 中，这样filter中提交的token就可以被处理了。

带Token请求校验流程

用户除登录之外的请求，都要求必须携带JWT Token。所以我们需要另外一个Filter对这些请求做一个拦截。这个拦截器主要是提取header中的token，跟登录一样，提交给 `AuthenticationManager` 做检查。

Filter

```
public class JwtAuthenticationFilter extends OncePerRequestFilter{
    ...
    public JwtAuthenticationFilter() {
        //拦截header中带Authorization的请求
        this.requiresAuthenticationRequestMatcher = new RequestHeaderRequestMatcher("A
    }
```

```
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse re
            throws ServletException, IOException {
        //header没带token的，直接放过，因为部分url匿名用户也可以访问
        //如果需要不支持匿名用户的请求没带token，这里放过也没问题，因为SecurityContext中没有认证信
        if (!requiresAuthentication(request, response)) {
            filterChain.doFilter(request, response);
            return;
        }
        Authentication authResult = null;
        AuthenticationException failed = null;
        try {
            //从头中获取token并封装后提交给AuthenticationManager
            String token = getJwtToken(request);
            if(StringUtils.isNotBlank(token)) {
                JwtAuthenticationToken authToken = new JwtAuthenticationToken(JWT.deco
                authResult = this.getAuthenticationManager().authenticate(authToken);
            } else {  //如果token长度为0
                failed = new InsufficientAuthenticationException("JWT is Empty");
            }
        } catch(JWTDecodeException e) {
            logger.error("JWT format error", e);
            failed = new InsufficientAuthenticationException("JWT format error", faile
        }catch (InternalAuthenticationServiceException e) {
            logger.error(
                    "An internal error occurred while trying to authenticate the user.
                    failed);
            failed = e;
        }catch (AuthenticationException e) {
            // Authentication failed
            failed = e;
        }
        if(authResult != null) {   //token认证成功
            successfulAuthentication(request, response, filterChain, authResult);
        } else if(!permissiveRequest(request)){
            //token认证失败，并且这个request不在例外列表里，才会返回错误
            unsuccessfulAuthentication(request, response, failed);
            return;
        }
        filterChain.doFilter(request, response);
    }

    ...

    protected boolean requiresAuthentication(HttpServletRequest request,
            HttpServletResponse response) {
        return requiresAuthenticationRequestMatcher.matches(request);
    }

    protected boolean permissiveRequest(HttpServletRequest request) {
        if(permissiveRequestMatchers == null)
            return false;
        for(RequestMatcher permissiveMatcher : permissiveRequestMatchers) {
            if(permissiveMatcher.matches(request))
                return true;
        }
        return false;
    }
}
```

这个Filter的实现跟登录的Filter有几点区别：

- 经过这个Filter的请求，会继续过 `FilterChain` 中的其它Filter。因为跟登录请求不一样，token只是为了识别用户。
- 如果header中没有认证信息或者认证失败，还会判断请求的url是否强制认证的（通过 `permissiveRequest` 方法判断）。如果请求不是强制认证，也会放过，这种情况比如博客类应用匿名用户访问查看页面；比如登出操作，如果未登录用户点击登出，我们一般是不会报错的。

  其它逻辑跟登录一样，组装一个token提交给 `AuthenticationManager` 。

JwtAuthenticationProvider

```java
public class JwtAuthenticationProvider implements AuthenticationProvider{

    private JwtUserService userService;

    public JwtAuthenticationProvider(JwtUserService userService) {
        this.userService = userService;
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws Authentic
        DecodedJWT jwt = ((JwtAuthenticationToken)authentication).getToken();
        if(jwt.getExpiresAt().before(Calendar.getInstance().getTime()))
            throw new NonceExpiredException("Token expires");
        String username = jwt.getSubject();
        UserDetails user = userService.getUserLoginInfo(username);
        if(user == null || user.getPassword()==null)
            throw new NonceExpiredException("Token expires");
        String encryptSalt = user.getPassword();
        try {
            Algorithm algorithm = Algorithm.HMAC256(encryptSalt);
            JWTVerifier verifier = JWT.require(algorithm)
                    .withSubject(username)
                    .build();
            verifier.verify(jwt.getToken());
        } catch (Exception e) {
            throw new BadCredentialsException("JWT token verify fail", e);
        }
        //成功后返回认证信息，filter会将认证信息放入SecurityContext
        JwtAuthenticationToken token = new JwtAuthenticationToken(user, jwt, user.getA
        return token;
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.isAssignableFrom(JwtAuthenticationToken.class);
    }

}
```

认证结果Handler

如果token认证失败，并且不在permissive列表中话，就会调用FailHandler，这个Handler和登录行为一致，所以都使用 `HttpStatusLoginFailureHandler` 返回401错误。

token认证成功，在继续FilterChain中的其它Filter之前，我们先检查一下token是否需要刷新，刷新成功后会将新token放入header中。所以，新增一个 `JwtRefreshSuccessHandler` 来处理token认证成功的情况。

```java
public class JwtRefreshSuccessHandler implements AuthenticationSuccessHandler{

    private static final int tokenRefreshInterval = 300;  //刷新间隔5分钟

    private JwtUserService jwtUserService;

    public JwtRefreshSuccessHandler(JwtUserService jwtUserService) {
        this.jwtUserService = jwtUserService;
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletRespons
            Authentication authentication) throws IOException, ServletException {
        DecodedJWT jwt = ((JwtAuthenticationToken)authentication).getToken();
        boolean shouldRefresh = shouldTokenRefresh(jwt.getIssuedAt());
        if(shouldRefresh) {
            String newToken = jwtUserService.saveUserLoginInfo((UserDetails)authentica
            response.setHeader("Authorization", newToken);
        }
    }

    protected boolean shouldTokenRefresh(Date issueAt){
        LocalDateTime issueTime = LocalDateTime.ofInstant(issueAt.toInstant(), ZoneId.
        return LocalDateTime.now().minusSeconds(tokenRefreshInterval).isAfter(issueTim
    }

```

```
public class JwtLoginConfigurer<T extends JwtLoginConfigurer<T, B>, B extends HttpSecu

    private JwtAuthenticationFilter authFilter;

    public JwtLoginConfigurer() {
        this.authFilter = new JwtAuthenticationFilter();
    }

    @Override
    public void configure(B http) throws Exception {
        authFilter.setAuthenticationManager(http.getSharedObject(AuthenticationManager
        authFilter.setAuthenticationFailureHandler(new HttpStatusLoginFailureHandler()
        //将filter放到logoutFilter之前
        JwtAuthenticationFilter filter = postProcess(authFilter);
        http.addFilterBefore(filter, LogoutFilter.class);
    }
    //设置匿名用户可访问url
    public JwtLoginConfigurer<T, B> permissiveRequestUrls(String ... urls){
        authFilter.setPermissiveUrl(urls);
        return this;
    }

    public JwtLoginConfigurer<T, B> tokenValidSuccessHandler(AuthenticationSuccessHand
        authFilter.setAuthenticationSuccessHandler(successHandler);
        return this;
    }

}
```

配置集成

整个登录和无状态用户认证的流程都已经讲完了，现在我们需要吧spring security集成到我们的web项目中去。spring security和spring mvc做了很好的集成，一共只需要做两件事，给web配置类加上 `@EanbleWebSecurity` ，继承 `WebSecurityConfigurerAdapter` 定义个性化配置。

配置类WebSecurityConfig

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter{

    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/image/**").permitAll() //静态资源访问无需认证
                .antMatchers("/admin/**").hasAnyRole("ADMIN") //admin开头的请求，需要admi
                .antMatchers("/article/**").hasRole("USER") //需登陆才能访问的url
                .anyRequest().authenticated()  //默认其它的请求都需要认证，这里一定要添加
                .and()
            .csrf().disable()  //CRSF禁用，因为不使用session
            .sessionManagement().disable()  //禁用session
            .formLogin().disable() //禁用form登录
            .cors()  //支持跨域
            .and()   //添加header设置，支持跨域和ajax请求
            .headers().addHeaderWriter(new StaticHeadersWriter(Arrays.asList(
                    new Header("Access-control-Allow-Origin","*"),
                    new Header("Access-Control-Expose-Headers","Authorization"))))
            .and() //拦截OPTIONS请求，直接返回header
            .addFilterAfter(new OptionRequestFilter(), CorsFilter.class)
            //添加登录filter
            .apply(new JsonLoginConfigurer<>()).loginSuccessHandler(jsonLoginSuccessHa
            .and()
           //添加token的filter
            .apply(new JwtLoginConfigurer<>()).tokenValidSuccessHandler(jwtRefreshSucc
            .and()
            //使用默认的logoutFilter
            .logout()
//              .logoutUrl("/logout")   //默认就是"/logout"
                .addLogoutHandler(tokenClearLogoutHandler())  //logout时清除token
                .logoutSuccessHandler(new HttpStatusReturningLogoutSuccessHandler()) /
            .and()
            .sessionManagement().disable();
    }
    //配置provider
```

```
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean("jwtAuthenticationProvider")
    protected AuthenticationProvider jwtAuthenticationProvider() {
        return new JwtAuthenticationProvider(jwtUserService());
    }

    @Bean("daoAuthenticationProvider")
    protected AuthenticationProvider daoAuthenticationProvider() throws Exception{
        //这里会默认使用BCryptPasswordEncoder比对加密后的密码，注意要跟createUser时保持一致
        DaoAuthenticationProvider daoProvider = new DaoAuthenticationProvider();
        daoProvider.setUserDetailsService(userDetailsService());
        return daoProvider;
    }
    ...
}
```

以上的配置类主要关注一下几个点：

- 访问权限配置，使用url匹配是放过还是需要角色和认证
- 跨域支持，这个我们下面再讲
- 禁用csrf，csrf攻击是针对使用session的情况，这里是不需要的，关于CSRF可参考 Cross Site Request Forgery
- 禁用默认的form登录支持
- logout支持，spring security已经默认支持logout filter，会拦截/logout请求，交给logoutHandler处理，同时在logout成功后调用 `LogoutSuccessHandler` 。对于logout，我们需要清除保存的token salt信息，这样再拿logout之前的token访问就会失败。请参考TokenClearLogoutHandler：

```
public class TokenClearLogoutHandler implements LogoutHandler {

    private JwtUserService jwtUserService;

    public TokenClearLogoutHandler(JwtUserService jwtUserService) {
        this.jwtUserService = jwtUserService;
    }

    @Override
    public void logout(HttpServletRequest request, HttpServletResponse response, Authe
        clearToken(authentication);
    }

    protected void clearToken(Authentication authentication) {
        if(authentication == null)
            return;
        UserDetails user = (UserDetails)authentication.getPrincipal();
        if(user!=null && user.getUsername()!=null)
            jwtUserService.deleteUserLoginInfo(user.getUsername());
    }

}
```

角色配置

Spring Security对于访问权限的检查主要是通过 `AbstractSecurityIntercepter` 来实现，进入这个拦截器的基础一定是在context有有效的Authentication。
回顾下上面实现的 `UserDetailsService` ，在登录或token认证时返回的 `Authentication` 包含了 `GrantedAuthority` 的列表。

```
@Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExce
```

admin和manager权限的用户访问：

```
.antMatchers("/admin/**").hasAnyRole("ADMIN,MANAGER")
```

对于Intecepter来说只需要吧配置中的信息和 `GrantedAuthority` 的信息一起提交给 `AccessDecisionManager` 来做比对。

跨域支持

前后端分离的项目需要支持跨域请求，需要做下面的配置。

CORS配置

首先需要在HttpSecurity配置中启用cors支持

```
http.cors()
```

这样spring security就会从 `CorsConfigurationSource` 中取跨域配置，所以我们需要定义一个Bean:

```
@Bean
    protected CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("*"));
        configuration.setAllowedMethods(Arrays.asList("GET","POST","HEAD", "OPTION"));
        configuration.setAllowedHeaders(Arrays.asList("*"));
        configuration.addExposedHeader("Authorization");
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource()
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
```

Header配置

对于返回给浏览器的Response的Header也需要添加跨域配置：

```
http..headers().addHeaderWriter(new StaticHeadersWriter(Arrays.asList(
         //支持所有源的访问
        new Header("Access-control-Allow-Origin","*"),
         //使ajax请求能够取到header中的jwt token信息
        new Header("Access-Control-Expose-Headers","Authorization"))))
```

OPTIONS请求配置

对于ajax的跨域请求，浏览器在发送真实请求之前，会向服务端发送OPTIONS请求，看服务端是否支持。对于options请求我们只需要返回header，不需要再进其它的filter，所以我们加了一个 `OptionsRequestFilter` ，填充header后就直接返回：

```
public class OptionsRequestFilter extends OncePerRequestFilter{

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse re
            throws ServletException, IOException {
        if(request.getMethod().equals("OPTIONS")) {
            response.setHeader("Access-Control-Allow-Methods", "GET,POST,OPTIONS,HEAD"
            response.setHeader("Access-Control-Allow-Headers", response.getHeader("Acc
            return;
        }
        filterChain.doFilter(request, response);
    }

}
```

总结

security这个名字。

所以这两个框架的选择问题就相对简单了：

1）如果系统中本来使用了spring，那优先选择spring security；

2）如果是web系统，spring security提供了更多的安全性支持

3）除次之外可以选择shiro

文章内使用的源码已经放在git上：Spring Security and JWT demo

[参考资料]

Spring Security Reference

空挡 奋斗中的80后

总资产24 共写了9.8W字 获得618个赞 共525个粉丝

关注

全部评论 36 只看作者 按时间倒序 按时间正序

Java雏鸡开发

30楼 2020.12.28 13:55

好文

赞 回复

卡文迪雨

29楼 2020.12.20 18:45

您好，有幸看到您的文章

我在loadUserByUsername加上数据库的查询，后总会报

MyUsernamePasswordAuthenticationFilter : An internal error occurred while trying to authenticate the user.

org.springframework.security.authentication.InternalAuthenticationServiceException:

空挡 作者
2020.12.20 23:23

你把daoAuthenticationProvider()方法中的daoProvider.setUserDetailsService(userDetailsService());改成daoProvider.setUserDetailsService(jwtUserService());试一下，好像是Bean定义的问题

回复

添加新评论

耐人寻味_76b3
28楼 2020.11.16 15:46

用户登出 又走了一次生成token的方法这个没啥意义吧

赞 回复

耐人寻味_76b3
27楼 2020.11.13 14:15

把代码拉下来看了一下 太强了！

赞 回复

Duskry
26楼 2020.10.09 10:35

很清楚

赞 回复

一条狗_99b5
25楼 2020.09.24 15:08

后半段看的有点蒙，得看看demo源码 再来一遍

赞 回复

8b5712db38bd
24楼 2020.07.16 11:00

好文章，之前也看了楼主写的shiro

赞 回复

1bbf0a5de8fa
23楼 2020.07.15 20:04

第一次这么认真的看完了一篇文章,注释和代码都提供了,真心感谢博主的付出,特意登入失传已久的账号给你点个赞...继续加油.

赞 回复

Ashin10
20楼 2020.06.17 18:12

禁用CSRF
有些地方写的真的是详细，我都不知道你怎么研究出来的

赞 回复

b11cf422eff7
19楼 2020.04.08 16:11

思路清晰，自心博主，赞

被以下专题收入，发现更多相似内容

security security Se security springs... SE JavaWeb

安全认证 展开更多

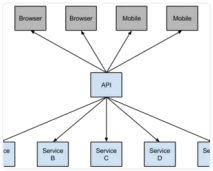
推荐阅读

更多精彩内容

Spring Cloud

Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智...

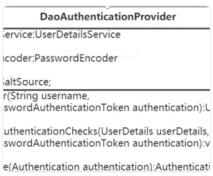
卡卡罗2017 阅读 122,385 评论 17 赞 134



Spring Security

参考文章 spring security 极客学院spring security 博客园Spring securi...

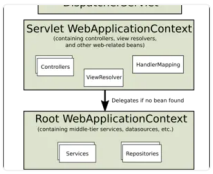
spilledyear 阅读 1,787 评论 0 赞 7



Spring Framework 5 MVC 官方手册译文

Spring Web MVC Spring Web MVC 是包含在 Spring 框架中的 Web 框架，建立于...

Hsinwong 阅读 18,093 评论 1 赞 89



spring boot 限制初始值大小及参数中文详解

要加“m”说明是MB，否则就是KB了. -Xms：初始值 -Xmx：最大值 -Xmn：最小值 java -Xms1...

阿B和阿C 阅读 6,074 评论 0 赞 7

谁的青春不迷茫

石家庄这两日天气不好，阴雨连绵，忽然间整个人都变得感伤起来。回想起去年的这个时候我正在学校的操场上拍各种搞怪的毕...

s狼烟s 阅读 208 评论 1 赞 3