链滴

Security 与响应式 webFlux(三) 完结撒花

作者：hong1yuan
原文链接：https://ld246.com/article/1599322291816
来源网站：链滴

前言

之前已经讲过WebFlux与Security的一种结合方式，又因为业务原因放弃了第一种实现的方式。

为了动态的**RBAC（Role-Based Access Control）** 和**接口级的权限管理**和利用Webflux的吞吐和应能力，又写另一种方式，这种方式放弃了一部分Security的能力，而利用自身的能力去书写，所以加灵活，**重要的是**，已经把该踩的坑已经全部踩平，如有新坑，还望互相交流。

实践篇

Config配置

之前已经写过，第一步需要配置Scurity的Config，跟之前有一些区别

```
@EnableWebFluxSecurity
@EnableReactiveMethodSecurity
public class SecurityConfig {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private SecurityContextRepository securityContextRepository;


    //security的鉴权排除列表
    private static final String[] excludedAuthPages = {
            "/auth/login",
            "/auth/logout"
    };
```

```
    @Bean
    SecurityWebFilterChain webFluxSecurityFilterChain(ServerHttpSecurity http) throws Excepti
n {

        return http
        .exceptionHandling()
                .authenticationEntryPoint((swe, e) -> {
                    return Mono.fromRunnable(() -> {
                        swe.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
                    });
                }).accessDeniedHandler((swe, e) -> {
                    return Mono.fromRunnable(() -> {
                        swe.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
                    });
                }).and()
                .csrf().disable()
                .formLogin().disable()
                .httpBasic().disable()
                .authenticationManager(authenticationManager)
                .securityContextRepository(securityContextRepository)
                .authorizeExchange()
                .pathMatchers(HttpMethod.OPTIONS).permitAll()
                .pathMatchers(excludedAuthPages).permitAll()
                .anyExchange().authenticated()
                .and().build();
    }
}
```

从上往下开始说明区别

1.增加认证异常处理，之前有登录成功与失败的类，现在已经不需要了，直接在配置遇到异常的处理式

.authenticationEntryPoint : 认证失败进行**HTTP 401状态码**返回

.accessDeniedHandler : 访问被拒绝进行**HTTP 403状态码**返回

.formLogin : Security登录认证功能关闭

.httpBasic : httpBasic功能关闭

.authenticationManage : 重写认证管理，并进行配置

.securityContextRepository : 重写Security上下文存储库并进行配置（这个在上章提到过）

省下的配置都已经讲过了，就不多说了，配置根据自身业务需要在进行修改，有很多功能。

securityContextRepository类

重写方法

```
@Component
public class SecurityContextRepository implements ServerSecurityContextRepository {

    @Value("${jwt.sign}")
```

```
    private String sign;


    @Override
    public Mono<Void> save(ServerWebExchange swe, SecurityContext sc) {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public Mono<SecurityContext> load(ServerWebExchange swe) {
        ServerHttpRequest request = swe.getRequest();
        String authHeader = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String authToken = authHeader.substring(7);
            User user = JwtUtil.verifyUserJwtToken(authToken, sign);
            if (authUser == null || authUser.getType() == null) {
                return Mono.empty();
            }
    //动态RBAC认证  伪代码 业务代码已删

//                  AtomicBoolean passAuthentication = new AtomicBoolean(false);
//                  user.getPermission().stream().forEach(permission ->{
//                      PathPattern pattern=new PathPatternParser().parse(permission.getPermission
));
//                      if (pattern.matches(request.getPath().pathWithinApplication()) && request.ge
MethodValue().equals(permission.getRequestType())){
//                          passAuthentication.set(true);
//                      }
//                  });
//                  if(!passAuthentication.get()){
//                      return Mono.empty();
//                  }

            Authentication auth = new UsernamePasswordAuthenticationToken(authToken, authT
ken);
            return this.authenticationManager.authenticate(auth).map((authentication) -> {
                return new SecurityContextImpl(authentication);
            });
        } else {
            return Mono.empty();
        }
    }

}
```

通过JWT方式解析出自己的token 利用token自己去通过redis等方式换取自己接口权限能力，在利用求路径中的接口路径来进行对比，如果权限认证失败就mono.empty();

在往常的代码中，咱们使用Secuirty的动态RBAC或**路径匹配**上，通常用**AntPathRequestMatcher**，在WebFlux中,虽然也可以使用此方法，但方法匹配中需要的**HttpServletRequest**则提供不了，所在WebFlux源码中，看到WebFlux中都是在使用**PathPattern**进行路径匹配，如果有更好的方式记得我。

AuthenticationManage类

```
@Component
@Slf4j
public class AuthenticationManager implements ReactiveAuthenticationManager {

    @Value("${jwt.sign}")
    private String sign;


    @Override
    public Mono<Authentication> authenticate(Authentication authentication) {
        String authToken = authentication.getCredentials().toString();

        try {
            User user = JwtUtil.verifyUserJwtToken(authToken, sign);
            List<GrantedAuthority> authorities = new ArrayList<>();
            authorities.add(new SimpleGrantedAuthority(user.getRole));
            return Mono.just(new UsernamePasswordAuthenticationToken(authentication, null, au
horities));
        } catch (Exception e) {
            return Mono.empty();
        }
    }

}
```

securityContextRepository类接收到token后就会进入认证类**AuthenticationManage类**

在这里，你可以把当前已登录的用户的token进行验证是否过期，token续期，角色和权限的赋予等，如果用户验证没问题 就可以**return Mono.just(new UsernamePasswordAuthenticationToke (authentication, null, authorities));** 进行认证通过给当前线程**赋予相应身份**,不通过就**return Mon .empty()** 进行401返回

token认证总结

上述就是当用户已经登录有Token该如何鉴权，如何通过Security认证的方案了，里面有些地方 需要据自身业务去进行完善，接下如何获取token就比较简单了 大致讲一下 就是通过接口 自己比较认证认证成功后根据jwt生成token返回给前端

Controller 获取token

跟正常MVC一样 可以自己写controller层 service serviceImpl层等，因为是gateway 获取库的数据以基于底层服务获取，调用feign或Http请求等方式获取用户信息，当然也可以直接连数据库进行调用

```
@RestController
public class AuthenticationController {

    @PostMapping("/login")
    public Mono<Result<TokenVO>> login(@RequestBody AuthVO authVO) {
       UserDtail user =userFeignClient.Detail(AuthVO);
        if (user!= null && new BCryptPasswordEncoder().matches(user.getPassword(),authVO.ge
Password()) ) {
    TokenVO tokenVO = new TokenVO();
    tokenVO.setToken(JwtUtil.createUserJwtToken(authUser,sign));
```

```
    }
    return Mono.just(Result.success(tokenVO));

}
```

通过查询用户数据后 根据密码查看是否正确，正确后生成token,因原版代码包含大量业务逻辑等，所简化了几十倍，根据业务自己写吧。

feign的调用中的坑

因为Webflux的原因，feign调用会报错，需要httpConverts一下

```
@Configuration
public class HttpMsgConverConfig {
    @Bean
    KeyResolver userKeyResolver() {
        return exchange -> Mono.just(Objects.requireNonNull(exchange.getRequest().getRemot
Address()).getAddress().getHostAddress());
    }

    @Bean
    @ConditionalOnMissingBean
    public HttpMessageConverters messageConverters(ObjectProvider<HttpMessageConverte
<?>> converters) {
        return new HttpMessageConverters(converters.orderedStream().collect(Collectors.toList()
);
    }
}
```

这样就可以调用feign了

总结

根据Webflux与Security的结合 网上资料比较少，并且往往不能结合现在的动态RBAC与分布式系统在权限更加细粒化的需求的阶段，在以大型微服务为基础进行鉴权认证，保证速度、吞吐、响应为要，希望能帮助有同样困惑的人。

ps:在上个月2020年8月21号 新的oauth来了！！！

Authorization Server将替代**Spring Security OAuth**为 **Spring** 社区提供**OAuth2.0**授权服务器支。经过四个月的努力，**Spring Authorization Server**项目中的**OAuth2.0**授权服务器开发库正式发了第一个版本。

0.0.1版本已经发布，在之前Oauth2.0 比较难用的情况下，spring从放弃oauth到社区回归 到迎接新变化社区回归 到0.0.1的版本， 希望新版本的oauth系统 可以更加贴近现在服务变化所需要的东西。

之前的Oauth有很多不方便的地方，管理成本也比单纯的Security大，希望新版本能好用，之后放弃纯的Security迎接新的Oauth授权，咱们之后的新Oauth见。