Spring Boot & Security

Spring Boot是什么

Spring Boot 使您能轻松地创建独立的、生产级的、基于 Spring 且能直接运行的应用程序。我们对 Spring 平台和第三方库有自己的看法，所以您从一开始只会遇到极少的麻烦。

Pom依赖变少了

```xml
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.ray</groupId>
    <artifactId>security-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>security-demo</name>
    <description>Demo project for Spring Boot</description>
```

```xml
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.0.4.RELEASE</version>
        <relativePath>../../spring-boot-dependencies</relativePath>
    </parent>
    <artifactId>spring-boot-starter-parent</artifactId>
    <packaging>pom</packaging>
    <name>Spring Boot Starter Parent</name>
    <description>Parent pom providing dependency and plugin management for applications
        built with Maven</description>
    <url>https://projects.spring.io/spring-boot/#/spring-boot-starter-parent</url>
```

我们使用了SpringBoot之后，由于父工程有对版本的统一控制，所以大部分第三方包，我们无需关注版本，个别没有纳入SpringBoot管理的，才需要设置版本号

自动装配

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

https://juejin.im/entry/5b447cbbe51d45199566f752

自动化配置的核心就是提供实体bean的组装和初始化，对于本starter而言，我们主要组装的核心bean就是 Jedis 这个实体。

```
@Configuration
@ConditionalOnClass(Jedis.class)    // 存在Jedis这个类才装配当前类
@EnableConfigurationProperties(RedisProperties.class)
public class RedisAutoConfiguration {



    @Bean
    @ConditionalOnMissingBean   // 没有Jedis这个类才进行装配
    public Jedis jedis(RedisProperties redisProperties) {
        return new Jedis(redisProperties.getHost(), redisProperties.getPort());
    }
}
```

可以看到，我们这里通过@Bean注解组装了一个Jedis的Bean，外界只需要通过注入即可使用。

@Configuration: 声明当前类为一个配置类
@ConditionalOnClass(Jedis.class)：当存在Jedis这个Class时才装配当前配置类
@EnableConfigurationProperties(RedisProperties.class)：这是一个开启使用配置参数的注解，value值就是我们配置实体参数映射的ClassType，将配置实体作为配置来源。
@ConditionalOnMissingBean：当SpringIoc容器内不存在Jedis这个Bean的时候才进行装配，否则不装配。

这些注解是我们实现自动配置的关键，体现了spring的哲学:

约定优于配置

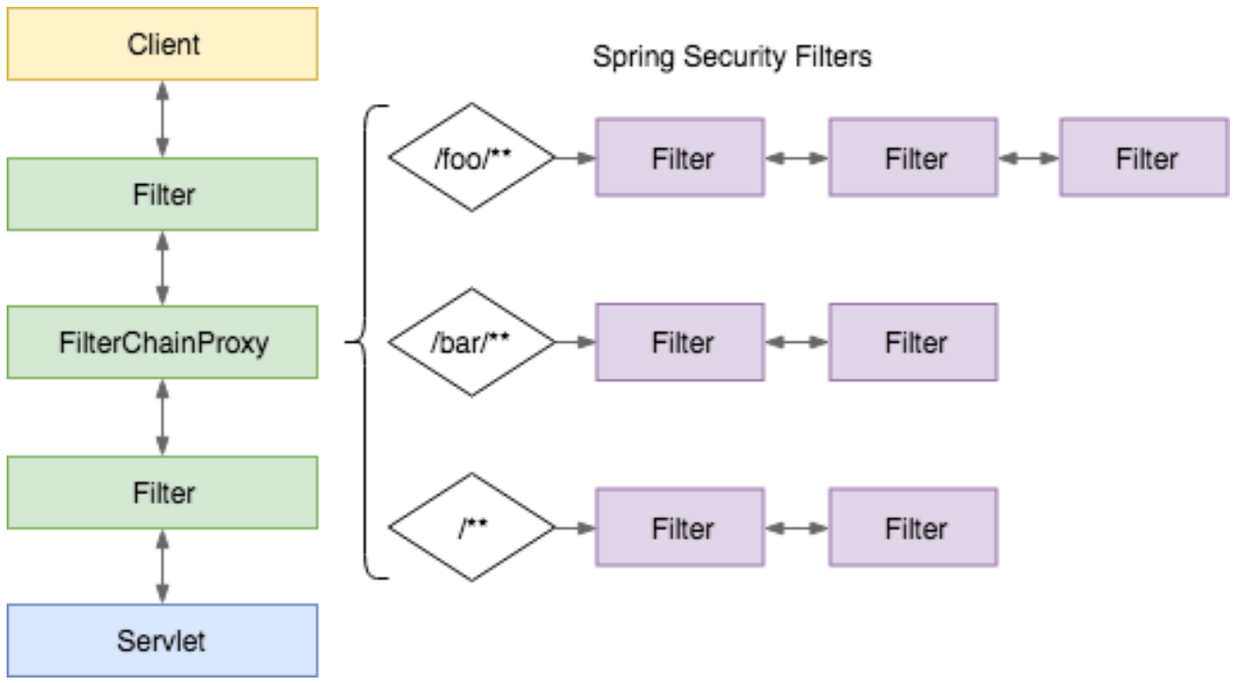
@ConditionalOnBean：当SpringIoc容器内存在指定Bean的条件
@ConditionalOnClass：当SpringIoc容器内存在指定Class的条件
@ConditionalOnExpression：基于SpEL表达式作为判断条件
@ConditionalOnJava：基于JVM版本作为判断条件
@ConditionalOnJndi：在JNDI存在时查找指定的位置
@ConditionalOnMissingBean：当SpringIoc容器内不存在指定Bean的条件
@ConditionalOnMissingClass：当SpringIoc容器内不存在指定Class的条件
@ConditionalOnNotWebApplication：当前项目不是Web项目的条件
@ConditionalOnProperty：指定的属性是否有指定的值
@ConditionalOnResource：类路径是否有指定的值
@ConditionalOnSingleCandidate：当指定Bean在SpringIoc容器内只有一个，
或者虽然有多个但是指定首选的Bean
@ConditionalOnWebApplication：当前项目是Web项目的条件

Spring Security

Spring Security是一个强大的和高度可定制的身份验证和访问控制框架。 它是保证基于spring的应用程序安全的实际标准。

基本原理

Java Servlet 和 Spring Security 都使用了设计模式中的责任链模式 。简单地说，它们都定义了许多过滤器（Filter），每一个请求都会经过层层过滤器的处理，最终返回。如下图：



Spring Security 在 Servlet 的过滤链（filter chain）中注册了一个过滤器 FilterChainProxy，它会把请求代理到 Spring Security 自己维护的多个过滤链，每个过滤链会匹配一些 URL，如图中的 /foo/**，如果匹配则执行对应的过滤器。过滤链是有顺序的，一个请求只会执行第一条匹配的过滤链。

Spring Security 的配置本质上就是新增、删除、修改过滤器。

权限认证，它其实分为两个部分

认证（Authentication）：即证明“你是你”，常见的如果用户名密码匹配，则认为操作者是该用户。

授权（Authorization）：即判断“你有没有资格”，例如“删贴”功能只允许管理员使用。

配置

```
@EnableWebSecurity
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)//打开方法级别的控制权限
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    /**
     * 配置可否访问的url
     * 请求匹配调度和授权
     */
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests() ExpressionInterceptUrlRegistry
                .antMatchers( ...antPatterns: "/", "/home", "/register") ExpressionUrlAuthorizationConfigurer<
                //.hasRole("ADMIN")  拥有admin
                .permitAll()  // / & /home 不需要认证
                .anyRequest() ExpressionUrlAuthorizationConfigurer<HttpSecurity>.AuthorizedUrl
                .authenticated() ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlRegistry

                .and() HttpSecurity
                .formLogin() FormLoginConfigurer<HttpSecurity>
                .loginPage("/login")          // 指定登录地址
                .permitAll() FormLoginConfigurer<HttpSecurity>

                .and() HttpSecurity
                .logout() LogoutConfigurer<HttpSecurity>
                .permitAll();
    }
```

继承WebSecurityConfigurerAdapter

重载configure方法

配置登录，登出等信息

配置不需要认证的url

配置需要某些权限的url

认证(Authentication)

一个从请求的报文中抽取用户名及密码信息等认证信息。认证信息需要实现 Authentication 接口。
另一个用来验证认证信息是否正确，如密码是否正确、API token 是否正确。
额外地，判断该用户是否有资格访问某个 URL，这个属于授权。

```java
/**
 * Processes an {@link Authentication} request.
 *
 * @author Ben Alex
 */
public interface AuthenticationManager {
    // ~ Methods
    // ========================================================================

    /**...*/
    Authentication authenticate(Authentication authentication)
            throws AuthenticationException;
}
```

如果认证通过，返回认证信息（比如擦除密码后的认证信息）
如果认证失败，抛 AuthenticationException 异常。
如果无法决定，返回 null。

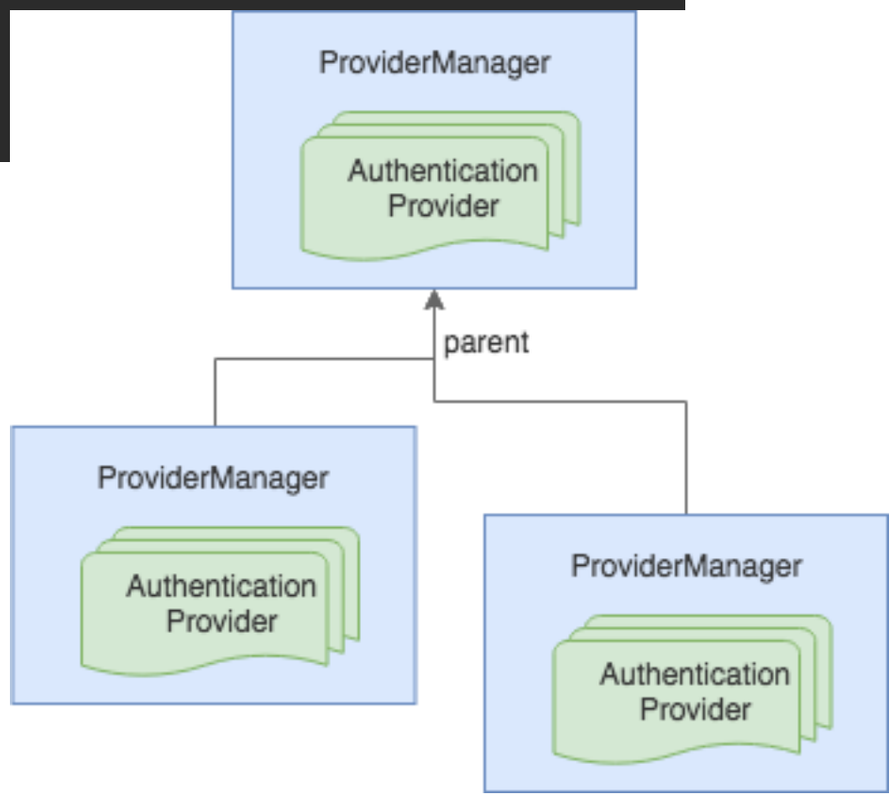
Spring Security 内部使用最多的实现是 ProviderManager，而它内部又使用了一个认证的链条，包含了多个AuthenticationProvier，ProviderManager 会逐一调用它们直到有一个 provider成功返回。

```java
/**
 * Indicates a class can process a specific
 * {@link org.springframework.security.core.Authentication} implementation.
 *
 * @author Ben Alex
 */
public interface AuthenticationProvider {
    // ~ Methods
    // ========================================================================

    /**...*/
    Authentication authenticate(Authentication authentication)
            throws AuthenticationException;

    /**...*/
    boolean supports(Class<?> authentication);
}
```

与 AuthenticationManager 不同的是它多了一个 supports 方法用来判断Provider 是否支持当前的认证信息。如一个 API Token 的认证器就不支持用户名密码的认证信息。



授权(Authorization)

要判断“你有没有资格”，首先要知道关于“你”的信息，也就是前一小节中说的 Authentication 接口；其次需要知道要访问的资源及资源的配置，如要访问 URL，该 URL 能被什么角色访问。类似地，Spring Security 已经定义了相关的接口，授权会在 FilterSecurityInterceptor 中启动。

```java
/**
 * Makes a final access control (authorization) decision.
 *
 * @author Ben Alex
 */
public interface AccessDecisionManager {
    //...

    /**...*/
    void decide(Authentication authentication, Object object,
            Collection<ConfigAttribute> configAttributes) throws AccessDeniedException,
            InsufficientAuthenticationException;

    /**...*/
    boolean supports(ConfigAttribute attribute);

    /**...*/
    boolean supports(Class<?> clazz);
}
```

函数 decide 会决定授权是否成功，如果权限不足则抛 AccessDeniedException 异常。

authentication 代表了“认证信息”，从中可以获得诸如当前用户的角色等信息

object 即要访问的资源，如某个 URL 或是某个函数

configAttributes 代表该资源的配置，如该 URL 只能被“管理员”角色（ROLE_ADMIN）访问。

Spring Security 中，具体的授权策略是“投票机制”，每一个 AccessDecisionVoter 都能投票，而最后如何统计结果，由 AccessDecisionManager 的具体实现决定。

AffirmativeBased 只需要有人赞成即可，默认实现

ConsensusBased 需要多数人赞成；

UnanimousBased 需要所有人赞成。默认使用 AffirmativeBased。

参考

https://spring.io/projects/spring-security

https://lotabout.me/2019/Token-Authentication-via-Spring-Security/