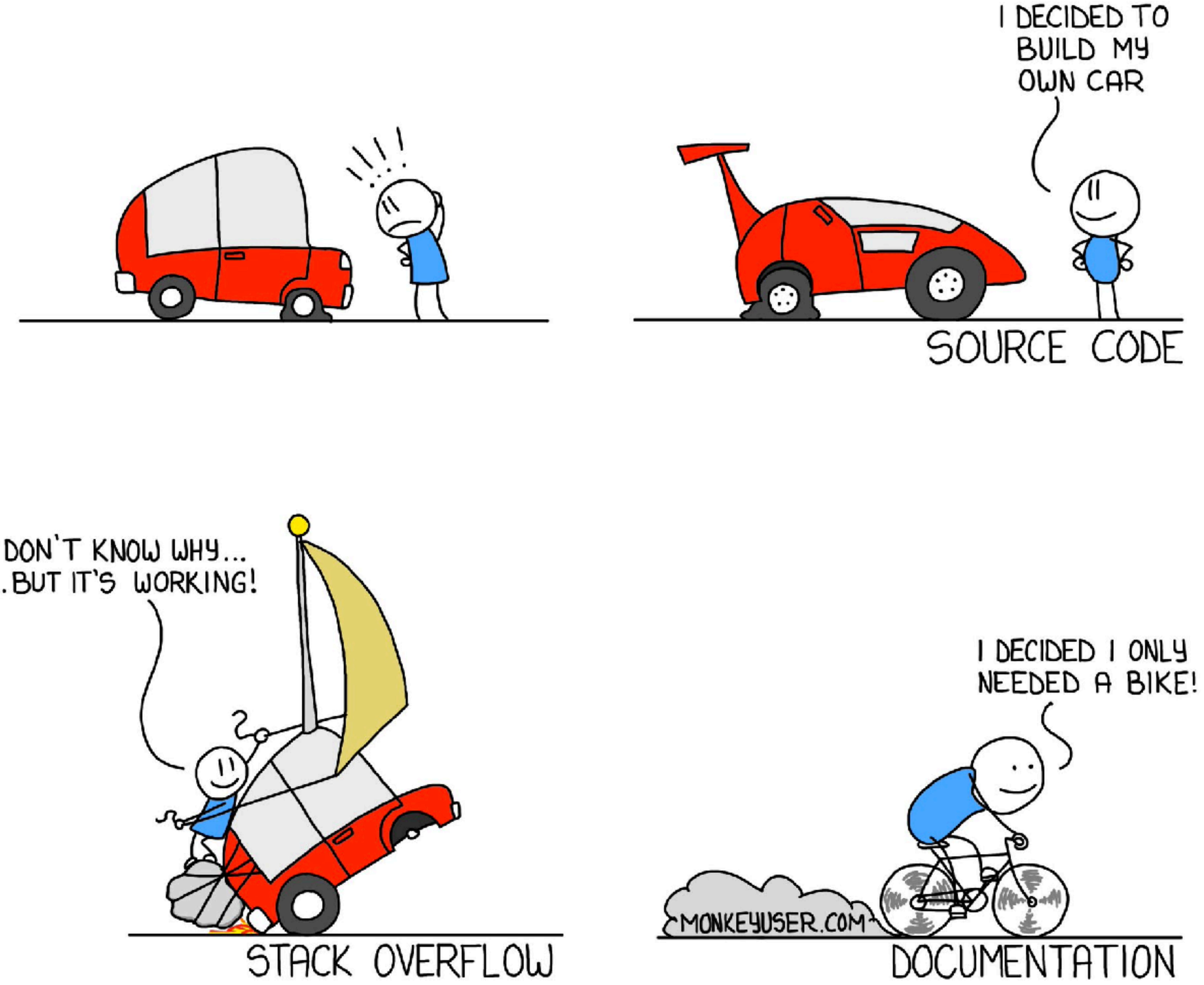
JAVA 达摩班

Spring Security

(BUG) FIXING CAR USING...


I DECIDED TO BUILD MY OWN CAR
SOURCE CODE
I DON'T KNOW WHY... ...BUT IT'S WORKING!
STACK OVERFLOW
I DECIDED I ONLY NEEDED A BIKE!
MONKEYUSER.COM
DOCUMENTATION

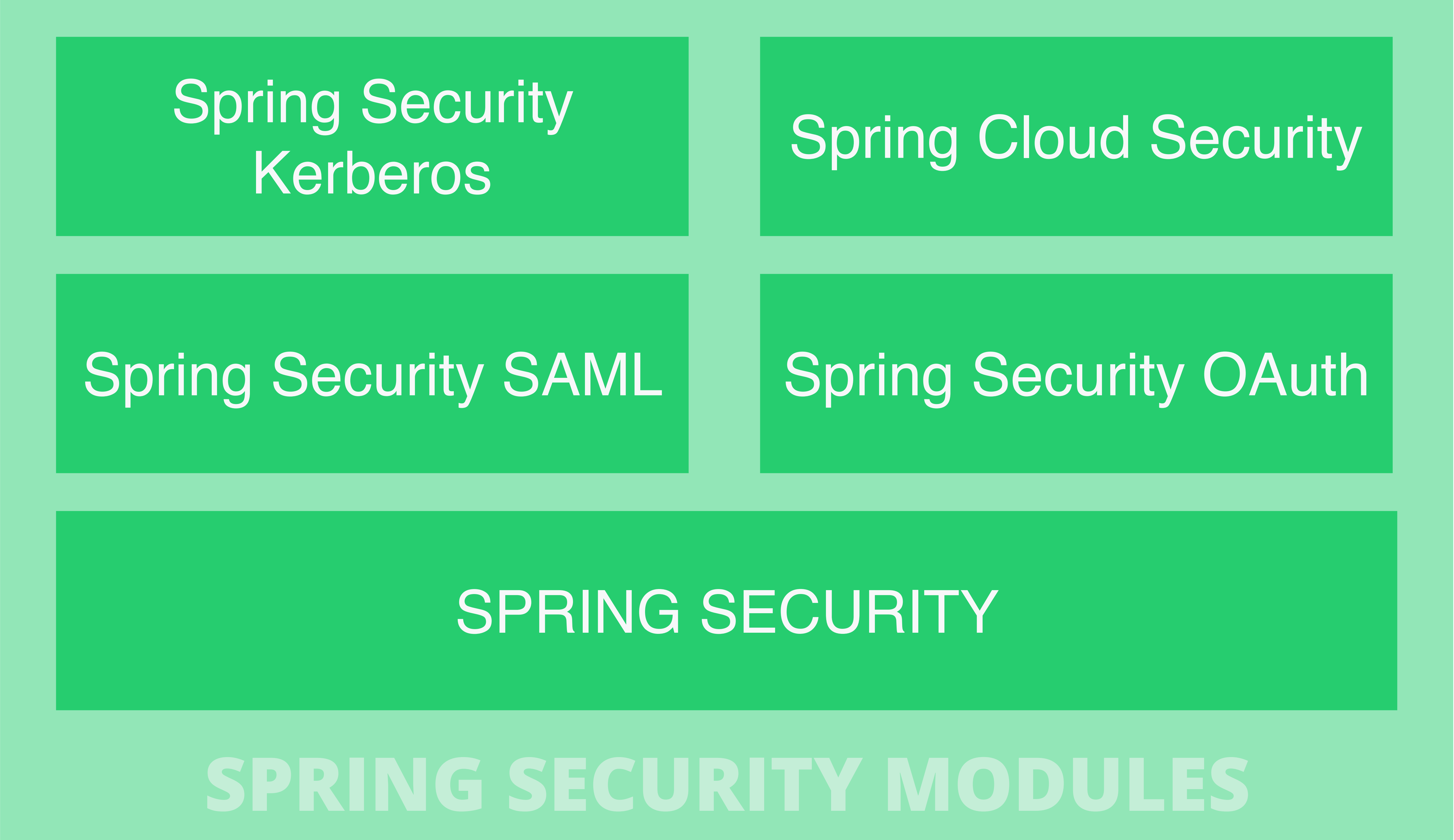
Spring Security 核心概念

1

Spring Security概念

最初Spring使用单独的第三方框架**Acegi Security**实现应用安全，但它需要大量XML配置，学习成本高以及不支持注释。Spring团队（Pivotal Team）将其并入Spring框架，成为Spring核心之一的“Spring Security”模块。

Spring 4安全模块如下：



Spring Security模块

Spring Security是Spring安全模块之一，是其他模块的基础。它是Java SE和Java EE的安全模块，用于实现Web应用或企业级应用的认证，授权，单点登陆等安全特性。

安全特性：（Spring 3.x）Authentication，Authorization，SSO (Single Sign-On)，Cross-Site Request Forgery (CSRF)，“Remember-Me”功能，ACL，“Channel Security”即HTTP和HTTPS切换，I18N，JAAS，Flow Authorization，WS-Security，XML配置和Annotations（Spring 4.x）WebSocket Security，Spring Data集成，CSRF Token

授权级别：**方法级别（通过AOP）**和 **URL级别（通过Servlet过滤器）**

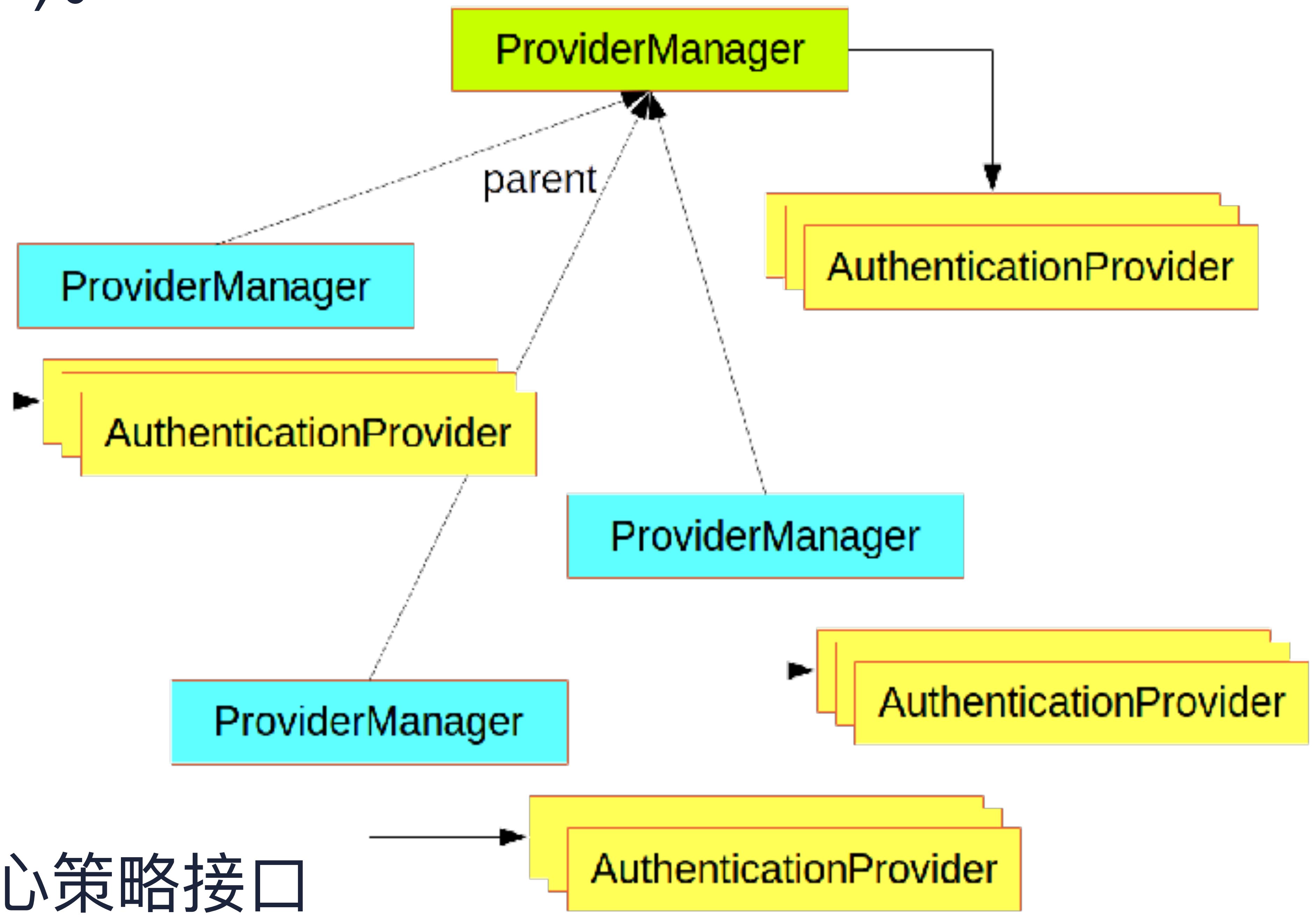
Spring Security概念

Spring Security为Java EE企业应用提供了全面的安全服务，它面向两个主要领域是**认证（authentication）**和**授权（访问控制，authorization/access-control）**。认证用于定义你是谁 (who are you?)，授权用于决定你能干嘛（what are you allowed to do?)。

- **认证和访问控制**

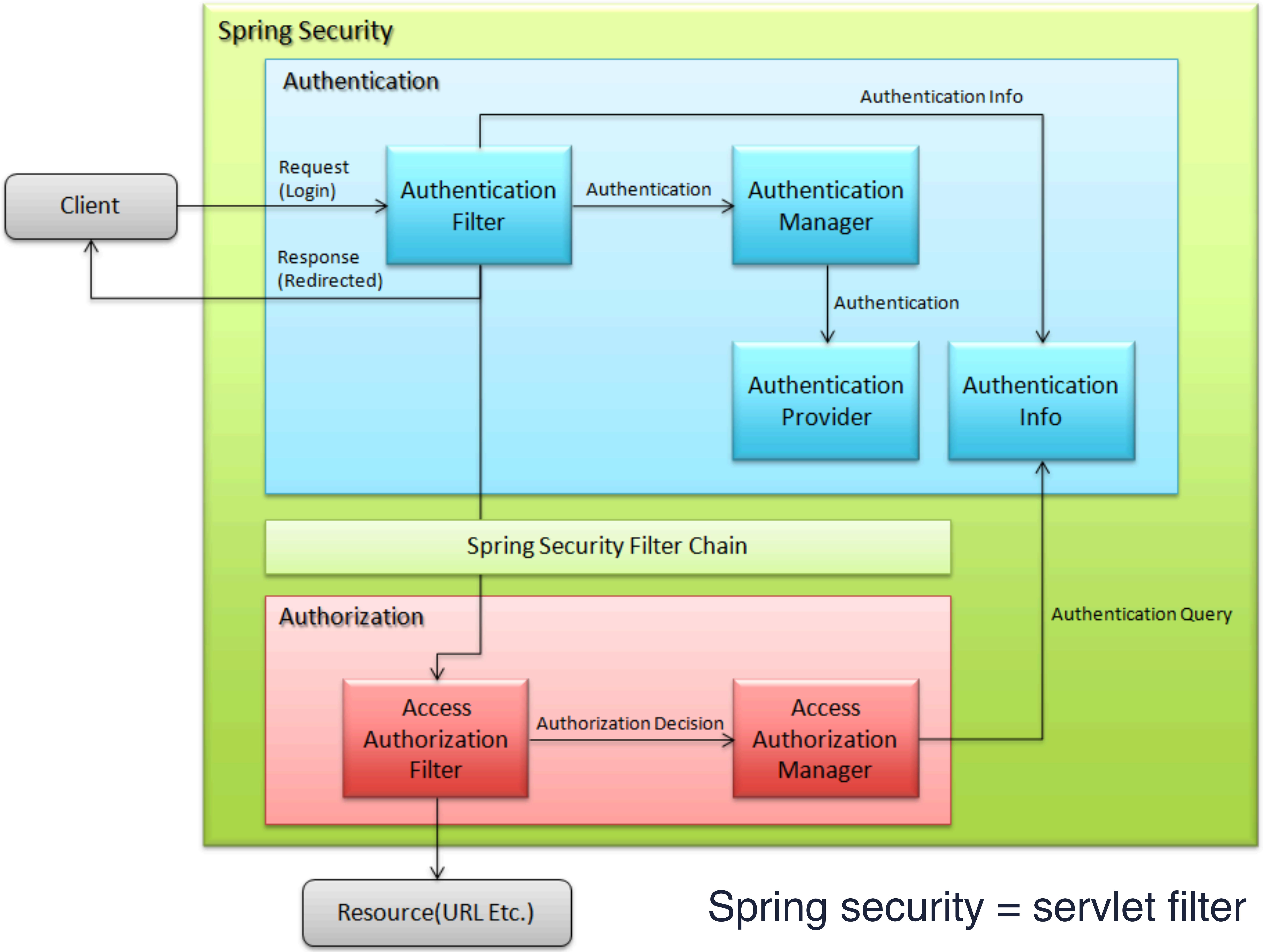
用于认证的主要接口是AuthenticationManager，它是函数式接口

```
public interface AuthenticationManager {
 Authentication authenticate(Authentication authentication)
   throws AuthenticationException;
}
```



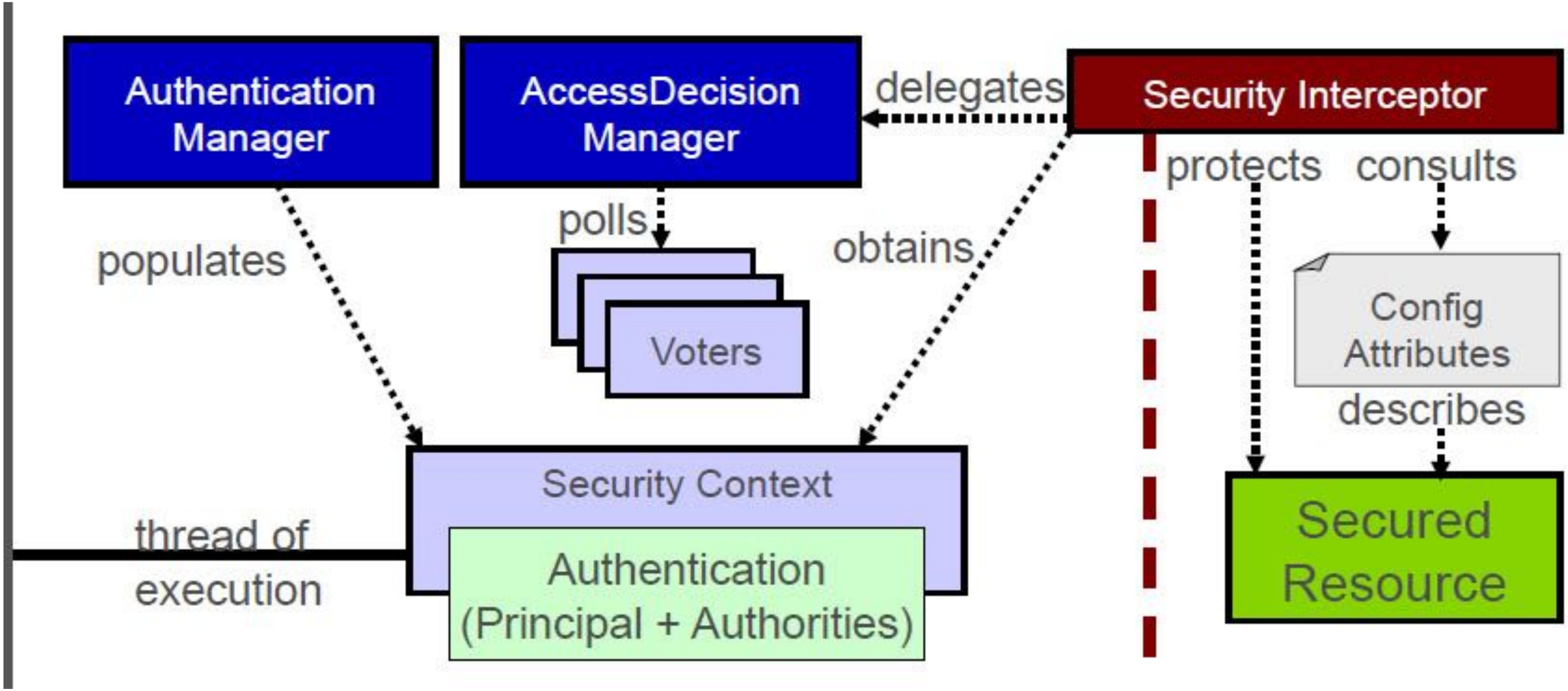
最常用的AuthenticationManager实现是ProviderManager，
它代理一串AuthenticationProvider实例。认证成功的下一步是授权，其核心策略接口是AccessDecisionManager，它的实现代理一串的AccessDecisionVoter。AccessDecisionVoter需要Authentication和Object对象，Object就代表用户想访问的资源（web资源，方法）

Spring Security概念



Spring security = servlet filter

Spring Security概念



Principal: 代表可以在应用中执行操作的用户，设备或其他系统

Spring Security概念

- **Web安全**

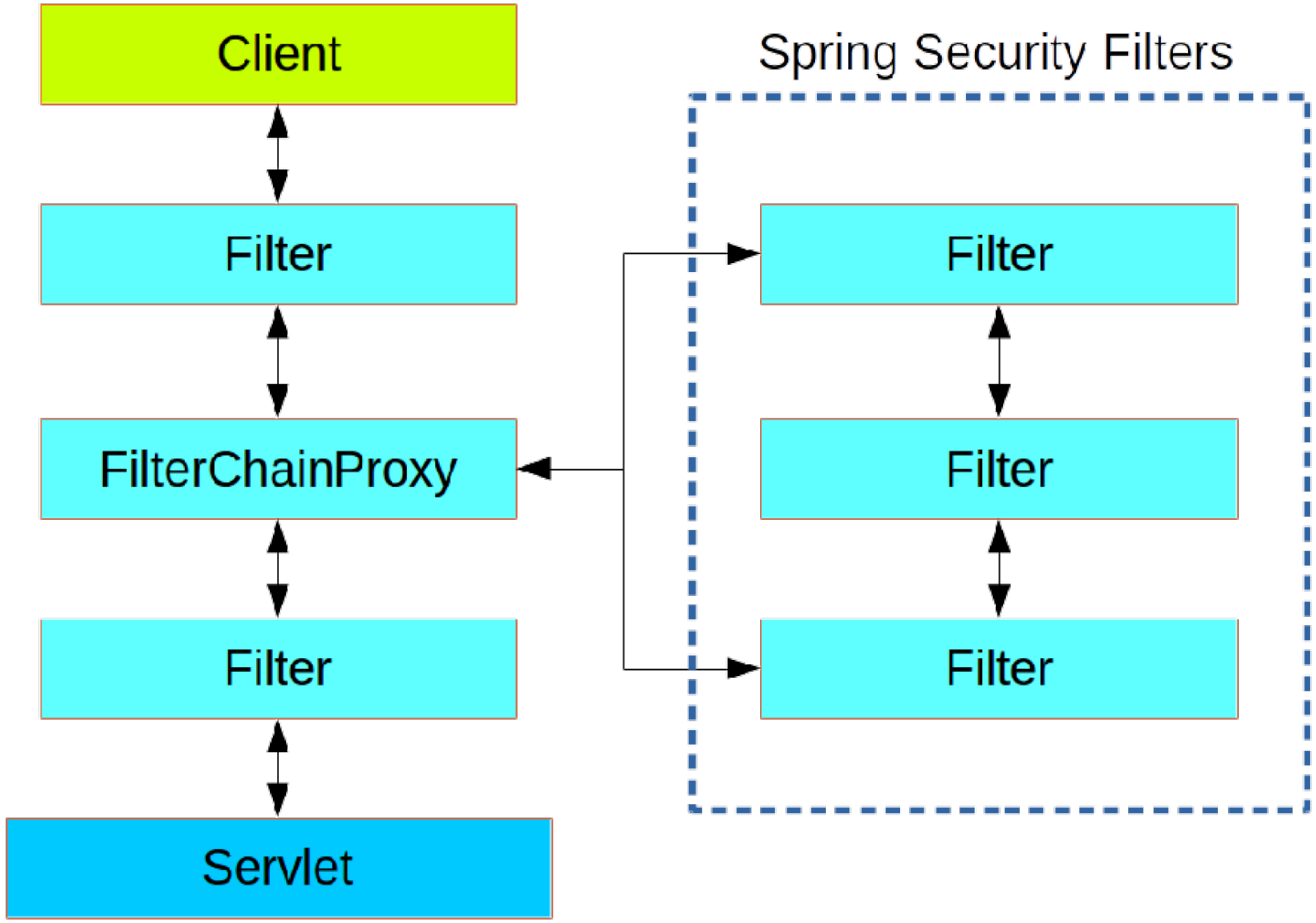
Spring security在web应用层面（UI和HTTP）是基于servlet filter的，不一样的是spring security作为一个单独的过滤器FilterChainProxy存在，内部包含多个过滤器，每个有自己的角色。在Spring boot中，它是ApplicationContext下的@Bean，并且默认接受所有请求。

- **方法安全**

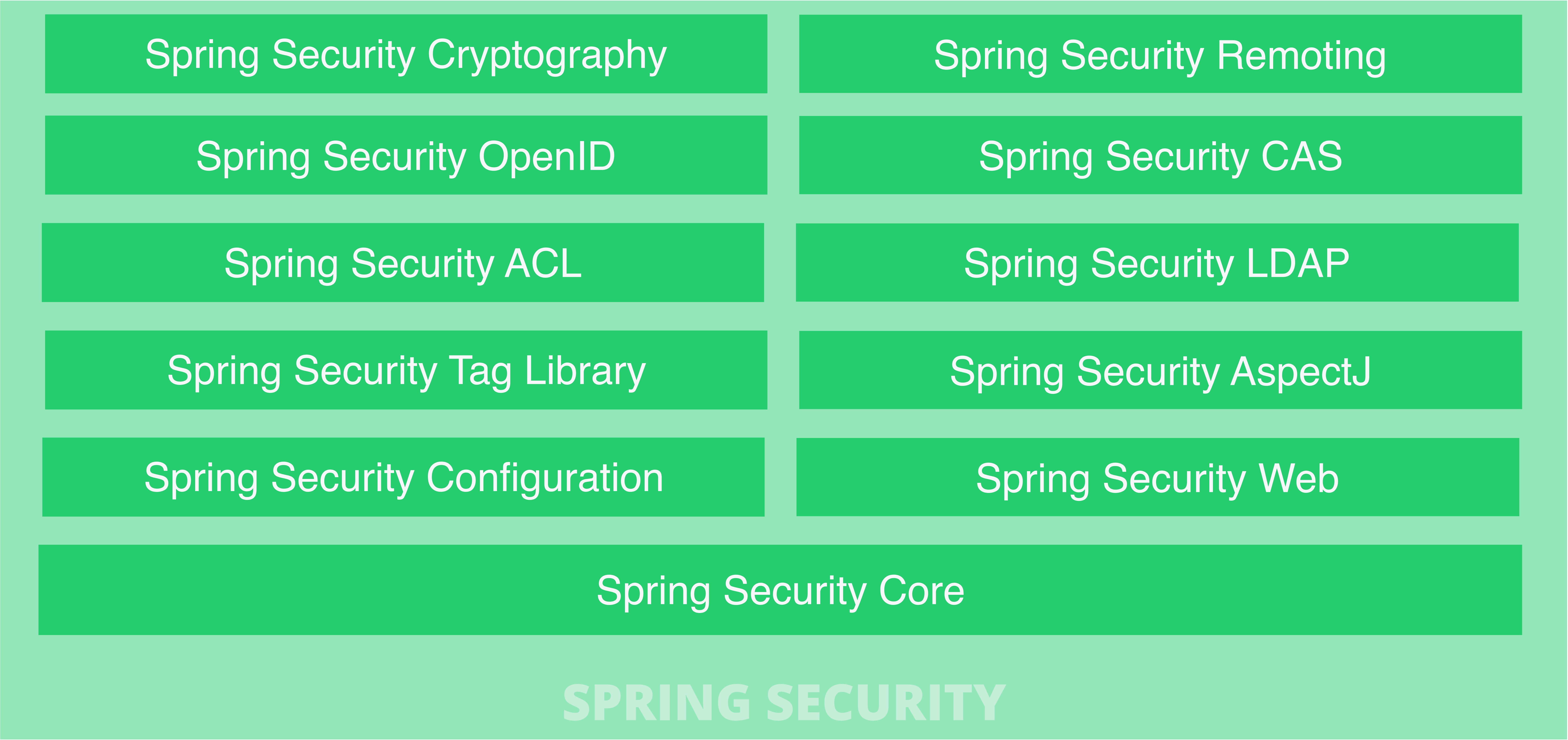
Spring security提供了精确到方法级别的访问控制，这里只是不同的“受保护资源”的类型。

- **线程绑定**

Spring Security是线程绑定的，因为它需要使下游用户可以访问并发认证当事人。基本构建模块是SecurityContext，它包含一个认证，当用户登陆后它就成为经过认证的认证。我们可以通过SecurityContextHolder中的静态方法访问和操作SecurityContext



Spring Security模块



Spring Security Cryptography

Spring Security Remoting

Spring Security OpenID

Spring Security CAS

Spring Security ACL

Spring Security LDAP

Spring Security Tag Library

Spring Security AspectJ

Spring Security Configuration

Spring Security Web

Spring Security Core

SPRING SECURITY

Security Maven依赖

Spring设计高度模块化，Spring Context并不依赖于Persistence或MVC
同理，Spring security也是，它是独立发布版本的

Spring和Spring security有各自的发布计划，不是1:1的版本号匹配，例如Spring security 3.1.x并于依赖于Spring 3.1.x，前者发布遭遇后者，但将来的发布会尽量匹配他们的版本。

spring-security-core是安全核心，包含认证和授权功能，它支持单独的应用（可以是非web应用）
spring-security-web是web应用安全，包含servlet环境下的过滤器实现和url访问控制

新老版本依赖问题

如果老版本spring security和新版本spring一起引用，spring security会引入老版本spring作为它的依赖。这和maven解决冲突原理有关，**maven会挑选距离依赖树根节点最近的jar包引入**。假设spring-orm依赖4.x.release，紧跟着spring-security-core依赖3.2.8.release，那么后面spring-jdbc会根据顺序取第一个作为自己的依赖
最佳实践是显式定义spring依赖，而不隐式依赖解决机制，也就是都处于pom的0层。

Spring Security 表达式

2

安全表达式

Spring security提供大量表达式，他们基于Spring Expression Language（SpEL）。大多数security表达式作用于上下文对象 - authenticated principal，由SecurityExpressionRoot执行，用于实现web和方法级别的安全。

SpEL在Spring Security 3.0引入， 4.x被放大。

Spring security提供两种Web授权类型：基于URL的正页面授权和基于安全规则的显示部分页面，通过使用hasRole表达式实现。

全页面授权

配置文件中通过intercept-url标签实现

```
<http use-expressions = "true">
    <intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')" />
    ...
</http>
```

Java配置中通过antMatchers和hasRole实现

Spring Security 4自动为角色增加前缀ROLE_

```
@Configuration
@EnableWebSecurity
public class SecSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
          .authorizeRequests()
          .antMatchers("/admin/**").hasRole("ADMIN");
    }
    ...
}
```

部分页面授权

- 首先确认spring-security-taglibs依赖正确引入

- 其次页面内增加taglib支持

```
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
```

- 最后根据条件和hasRole表达式控制某段html显示或隐藏

```
<security:authorize access="hasRole('ROLE_USER')">
   ...
</security:authorize>
<security:authorize access="hasRole('ROLE_ADMIN')">
   ...
</security:authorize>
```

方法级别授权

Spring 3以后使用@PreAuthorize和@PostAuthorize (以及@PreFilter and @PostFilter)实现业务代码授权，并且支持SqEL

xml配置通过***<global-method-security pre-post-annotations="enabled" />***启动方法授权
Java配置通过注释@EnableGlobalMethodSecurity实现

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    ...
}
```

方法授权通过PreAuthorize实现，如下代码表示只有Admin才能调用该方法。
Pre和Post注释都是由CGLIB代理解析，所以类和方法不能生命为final。

```
@Service
public class FooService {
   @PreAuthorize("hasRole('ROLE_ADMIN')")
   public List<Foo> findAll() { ... }
}
```

认证主体

authenticated principal

通过request得到**授权角色**

```
@RequestMapping
public void someControllerMethod(HttpServletRequest request) {
    request.isUserInRole("someAuthority");
}
```

除了在控制器内获取角色，Spring还提供多种方式能获取认证信息

1. Bean内静态方法

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
if (!(authentication instanceof AnonymousAuthenticationToken)) {
    String currentUserName = authentication.getName();
    return currentUserName;
}
```

这种方式降低测试能力，通过自定义门面接口实现更加常见，它会隐藏静态方法调用，调用时注入即可

认证主体

authenticated principal

```
public interface IAuthenticationFacade {
    Authentication getAuthentication();
}
@Component
public class AuthenticationFacade implements IAuthenticationFacade {
    @Override
    public Authentication getAuthentication() {
        return SecurityContextHolder.getContext().getAuthentication();
    }
}
```

2. 控制器内通过Principal或Authentication参数获取

```
@RequestMapping(value = "/username", method = RequestMethod.GET)
@ResponseBody
public String currentUserName(Principal principal) {
    return principal.getName();
}
```

认证主体

authenticated principal

Authentication是非常灵活的接口，提供多种获取认证的方式

```
  @RequestMapping(value = "/username", method = RequestMethod.GET)
  @ResponseBody
  public String currentUserName(Authentication authentication) {
      return authentication.getName();
  }
```

类似的

```
   @RequestMapping(value = "/username", method = RequestMethod.GET)
   @ResponseBody
   public String currentUserNameSimple(HttpServletRequest request) {
       Principal principal = request.getUserPrincipal();
       return principal.getName();
   }
}
```

认证主体

authenticated principal

甚至可以转换认证为*User*对象

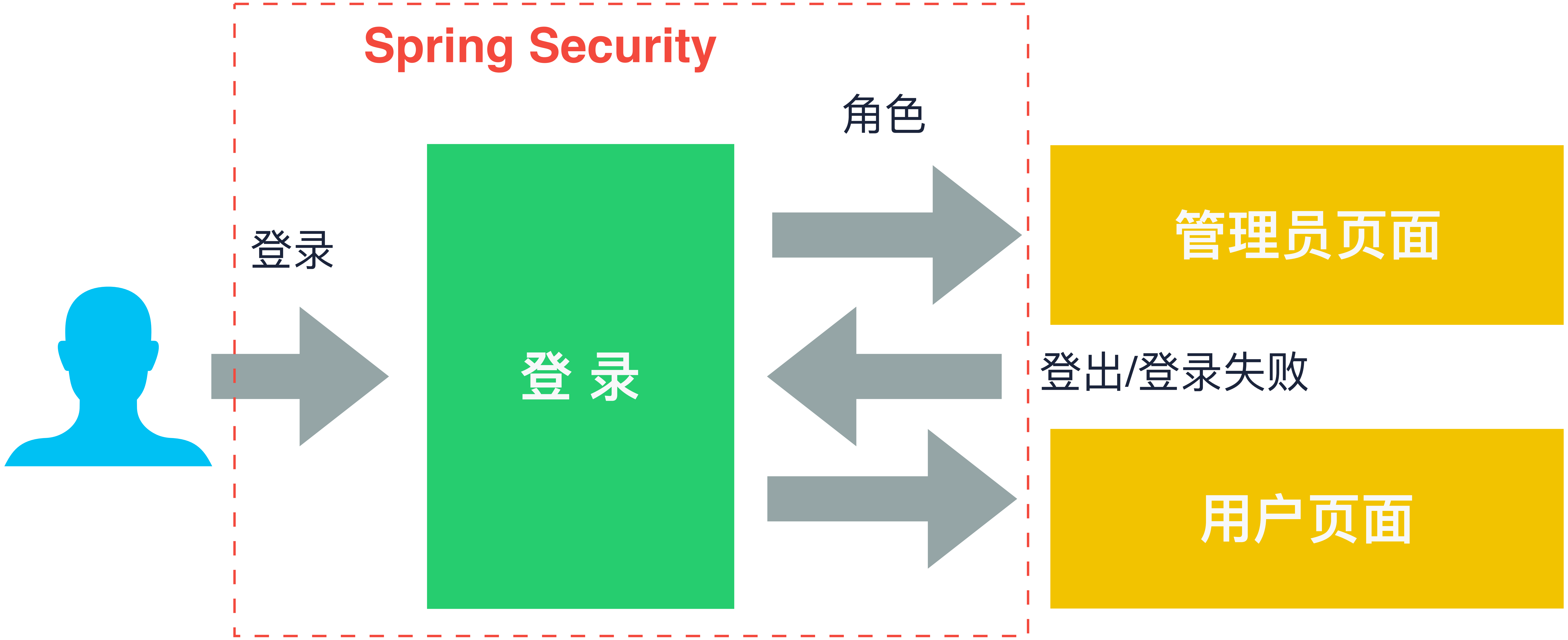
```
  UserDetails userDetails = (UserDetails) authentication.getPrincipal();
 System.out.println("User has authorities: " + userDetails.getAuthorities());
```

3. 通过安全标签实现JSP读取认证

```
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
...
<security:authorize access="isAuthenticated()">
    authenticated as <security:authentication property="principal.username" />
</security:authorize>
```

Spring Security 模拟用户登陆 3

登录流程



配置文件

web.xml增加如下内容，表示任何URL都要经过安全过滤器代理DelegatingFilterProxy，它会代理给FilterChainProxy，它会完整的管理bean生命周期

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

配置文件

Spring配置大部分通过Java configuration实现，但安全配置并不全部支持Java，很多需要XML文件配置。

```
@Configuration
@ImportResource({ "classpath:webSecurityConfig.xml" })
public class SecSecurityConfig {
   public SecSecurityConfig() {
      super();
   }
}
```

配置文件

<intercept-url>用于拦截URL，元素顺序很重要，特定规则在前，通用规则在后

```
<intercept-url pattern="/login*" access="isAnonymous()" />
<intercept-url pattern="/**" access="isAuthenticated()"/>
```

<form-login>用于定义登陆相关url，页面以及错误页面。

如果不指定login-page，Spring使用默认登录页面：/spring_security_login并生成默认表单；如果不指定请求URL，Spring使用默认url：/j_spring_security_check；如果不指定default-target-url，Spring默认跳到应用根目录。Spring安全配置推荐修改默认页面，以防止外部猜测出URL和页面地址。

```
<form-login
  login-page='/login.html'
  login-processing-url="/perform_login"
  default-target-url="/homepage.html"
  authentication-failure-url="/login.html?error=true"
  always-use-default-target="true"/>
```

配置文件

和form-login类似，logout-url用于指定登出url，触发登出机制，默认为/logout；logout-success-url用于指定登出成功后要跳转的页面，默认为/；success-handler-ref用于指定在等出成功后执行的方法，例如用于统计。success-handler-ref和logout-success-url同时只能实现一个。

登出页面还需要管理session和cookie，invalidate-session用于使session失效，默认为true；而delete-cookie用于删除cookie。

JSESSIONID cookie是J2EE应用cookie，用于追踪session，记录状态。它由web容器创建，随请求响应一并发回客户端

```
<logout
  logout-url="/perform_logout"
  delete-cookies="JSESSIONID"
  success-handler-ref="customLogoutSuccessHandler" />
...
<beans:bean name="customUrlLogoutSuccessHandler" />
```

配置文件

Authentication provider使用基于内存的InMemoryUserDetailsManager实现，不需要访问数据库，通过硬编码实现用户名和密码的验证（spring 3.1及以上）

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="user1" password="user1Pass" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

配置文件

与其对应的Java Configuration如下：

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
    .withUser("user1").password("user1Pass").roles("USER");
}

@Override
protected void configure(HttpSecurity http) throws Exception {
http
    .authorizeRequests().antMatchers("/admin/**").hasRole("ADMIN").antMatchers("/
anonymous*").anonymous().antMatchers("/login*").permitAll().anyRequest().authenticated()
    .and()
    .formLogin().loginPage("/login.html").loginProcessingUrl("/perform_login").defaultSuccessUrl("/
homepage.html",true).failureUrl("/login.html?error=true")
    .and()
    .logout().logoutUrl(“/
perform_logout").deleteCookies("JSESSIONID").logoutSuccessHandler(logoutSuccessHandler());
}
```

页面

登录页面通过表单form实现登录，action是配置中的登录URL，验证内容是表单中的用户名和密码字段

```
<form name='f' action="perform_login" method='POST'>
  <table>
     <tr>
        <td>User:</td>
        <td><input type='text' name='username' value=''></td>
     </tr>
     <tr>
        <td>Password:</td>
        <td><input type='password' name='password' /></td>
     </tr>
     <tr>
        <td><input name="submit" type="submit" value="submit" /></td>
     </tr>
  </table>
</form>
```

相应的登录成功页面的退出实现为

```
<a href="<c:url value="/perform_logout" />">Logout</a>
```

“记住我”

“remember me”机制用于跨session识别用户，“记住我”功能在session超时以后才起作用（可删除JSESSIONID，刷新页面，如果有“记住我”页面还在，否则跳回登录页面），默认30分钟以后。

首先，xml配置文件增加

```
<remember-me key="uniqueAndSecret"  remember-me-parameter="custom-remember-me" token-validity-seconds="86400"/>
```

key表示一个私有值，对整个应用是保密的，用于生成token内容，token有效期可配置，默认为两周

其次，login页面增加代码片段，name通常用remember-me，spring 3.2以后name字段可在xml中配置

```
<tr>
  <td>Remember Me:</td>
  <td><input type="checkbox" name="custom-remember-me" /></td>
</tr>
```

用户登录成功后，“记住我”会生成相应cookie，内容包括**username，expirationTime和MD5 hash**

MD5哈希值根据用户名，过期时间，密码和key生成

注意：该cookie被捕获存在安全风险，因为它是有效的，可用的

数据库初始化　4

基于JPA实现数据库初始化

JPA通过DDL（data definition language）数据定义语言生成，用于在程序启动时初始化数据库
初始化数据库的过程就是生成schema和进行数据操作的过程（DDL+DML）

spring.jpa.hibernate.ddl-auto属性有四个枚举：none, validate, update, create, 和create-drop。Spring boot会根据数据库类型设置不同的初始值（嵌入型为create-drop，其他为none），推荐自定义而不是用默认值。

classpath根目录下的import.sql，或data.sql文件会在程序启动时自动执行，schema-${platform}.sql中的paltform根据spring.datasource.platform配置而变化，用于适配不同的数据库类型

数据库迁移工具：Flyway和Liquibase

基于JPA实现数据库初始化

假设数据持久化使用JPA，并且定义如下Entity，Spring boot在启动应用时自动创建空表

```
@Entity
public class Country {
   @Id
   @GeneratedValue(strategy = IDENTITY)
   private Integer id;
   @Column(nullable = false)
   private String name;
}
```

如果创建如下内容的**data.sql**文件在classpath下，Spring会自动读取并插入表中

```
INSERT INTO country (name) VALUES ('India');
...
```

如果不想使用JPA默认建表，可以定义***schema.sql***文件，同样Spring会自动执行

```
CREATE TABLE country (
   id   INTEGER      NOT NULL AUTO_INCREMENT,
   name VARCHAR(128) NOT NULL,
   PRIMARY KEY (id)
);
```

同时关闭自动创建属性

```
spring.jpa.hibernate.ddl-auto=none
```

应用初始化

5

Spring应用初始化

为了更好的实现IOC，Spring将控制权更多下放给容器，所以对于实例化，或初始化逻辑需要特殊的处理方式，而不是直接写在Bean的构造函数中，或在实例化后调用特定方法。例如在构造函数中调用autowired字段，由于注入不一定完成，会出现NullPointerException

1. 通过@PostConstruct注解，它在bean初始化后立即执行

```
@Component
public class PostConstructExampleBean {
    private static final Logger LOG
      = Logger.getLogger(PostConstructExampleBean.class);

    @Autowired
    private Environment environment;

    @PostConstruct
    public void init() {
        LOG.info(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

Spring应用初始化

2. 实现InitializingBean接口，功能和上面完全一样

```
@Component
public class InitializingBeanExampleBean implements InitializingBean {
    private static final Logger LOG
      = Logger.getLogger(InitializingBeanExampleBean.class);

    @Autowired
    private Environment environment;

    @Override
    public void afterPropertiesSet() throws Exception {
        LOG.info(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

Spring应用初始化

3. XML init-method实现初始化

```
<bean id="initMethodExampleBean"
  class=“org.dharma.startup.InitMethodExampleBean"
  init-method="init">
</bean>

public class InitMethodExampleBean {
    private static final Logger LOG = Logger.getLogger(InitMethodExampleBean.class);

    @Autowired
    private Environment environment;

    public void init() {
        LOG.info(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

Spring应用初始化

4. 构造函数注入

```
@Component
public class LogicInConstructorExampleBean {
    private static final Logger LOG = Logger.getLogger(LogicInConstructorExampleBean.class);

    private final Environment environment;

    @Autowired
    public LogicInConstructorExampleBean(Environment environment) {
        this.environment = environment;
        LOG.info(Arrays.asList(environment.getDefaultProfiles()));
    }
}
```

Spring应用初始化

5. 实现ApplicationListener接口，它的逻辑在spring context（即所有bean）初始化完执行

```
@Component
public class StartupApplicationListenerExample implements
 ApplicationListener<ContextRefreshedEvent> {

  private static final Logger LOG
  = Logger.getLogger(StartupApplicationListenerExample.class);

  public static int counter;

  @Override
  public void onApplicationEvent(ContextRefreshedEvent event) {
    LOG.info("Increment counter");
    counter++;
  }
}
```

最新写法是使用@EventListener

```
@Component
public class EventListenerExampleBean {

  private static final Logger LOG
   = Logger.getLogger(EventListenerExampleBean.class);

  public static int counter;

  @EventListener
  public void onApplicationEvent(ContextRefreshedEvent event) {
    LOG.info("Increment counter");
    counter++;
  }
}
```

Spring应用初始化

6.Spring boot提供了CommandLineRunner实现和上面一样的功能

```
@Component
public class CommandLineAppStartupRunner implements CommandLineRunner {

    private static final Logger LOG = LoggerFactory.getLogger(CommandLineAppStartupRunner.class);

    public static int counter;

    @Override
    public void run(String...args) throws Exception {
        LOG.info("Increment counter");
        counter++;
    }
}
```

Spring应用初始化

7. Spring Boot还提供了ApplicationRunner接口实现同样功能，只是参数不同

```
@Component
public class AppStartupRunner implements ApplicationRunner {
    private static final Logger LOG = LoggerFactory.getLogger(AppStartupRunner.class);

    public static int counter;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        LOG.info("Application started with option names : {}", args.getOptionNames());
        LOG.info("Increment counter");
        counter++;
    }
}
```

综合应用

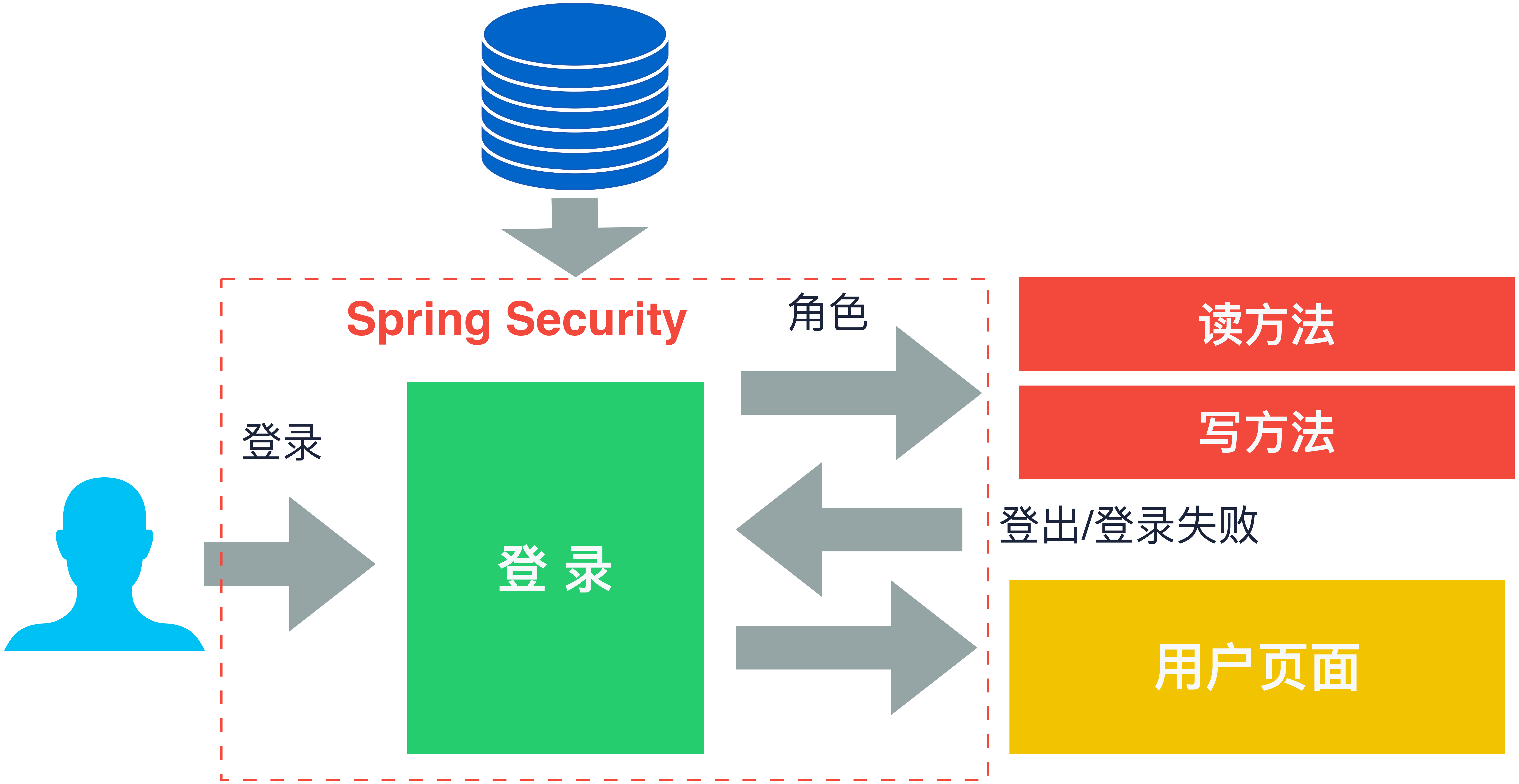
```
@Component
@Scope(value = "prototype")
public class AllStrategiesExampleBean implements InitializingBean {
    private static final Logger LOG = Logger.getLogger(AllStrategiesExampleBean.class);

    public AllStrategiesExampleBean() {
        LOG.info("Constructor");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        LOG.info("InitializingBean");
    }

    @PostConstruct
    public void postConstruct() {
        LOG.info("PostConstruct");
    }

    public void init() {
        LOG.info("init-method");
    }
}
```

```
[main] INFO o.b.startup.AllStrategiesExampleBean - Constructor
[main] INFO o.b.startup.AllStrategiesExampleBean - PostConstruct
[main] INFO o.b.startup.AllStrategiesExampleBean - InitializingBean
[main] INFO o.b.startup.AllStrategiesExampleBean - init-method
```

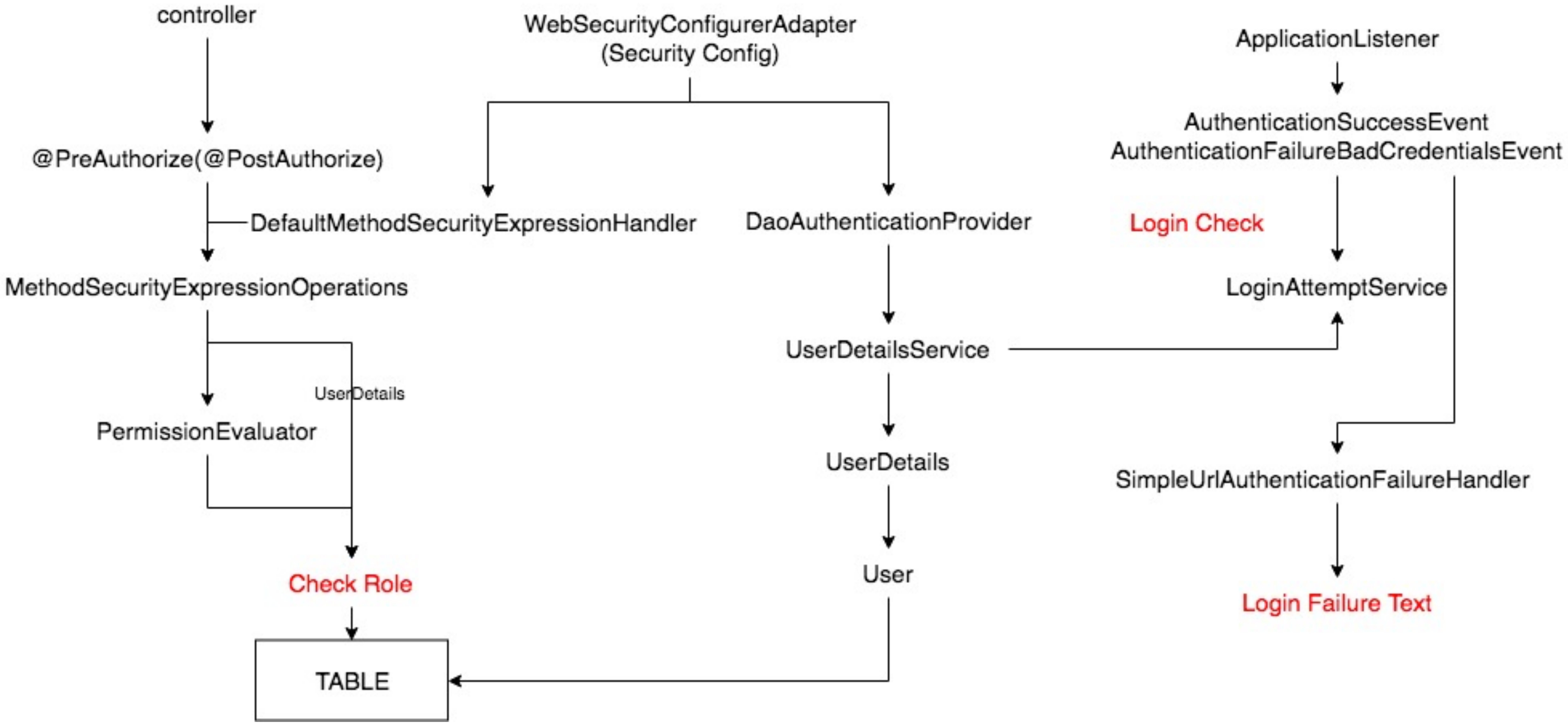
Spring Security + MySQL 实现认证授权

6

认证 + 授权



Spring MVC
Spring Security



controller

@PreAuthorize(@PostAuthorize)

DefaultMethodSecurityExpressionHandler

MethodSecurityExpressionOperations

UserDetails

PermissionEvaluator

Check Role

TABLE

WebSecurityConfigurerAdapter
(Security Config)

DaoAuthenticationProvider

UserDetailsService

UserDetails

User

ApplicationListener

AuthenticationSuccessEvent
AuthenticationFailureBadCredentialsEvent

Login Check

LoginAttemptService

SimpleUrlAuthenticationFailureHandler

Login Failure Text

用户验证

UserDetailsService接口用于获取用户数据，它包含一个方法loadUserByUsername用于通过用户名获取user实体，它可以用于重写来实现对特定用户对象的获取

DaoAuthenticationProvider用它在认证过程中加载用户信息

1. 定义User实体类

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    private String password;

    //standard getters and setters
}
```

用户验证

2. 定义User JPA用于操作用户数据

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

3. 实现UserDetails接口，用户封装用户信息

```
public class MyUserPrincipal implements UserDetails {
    private User user;

    public MyUserPrincipal(User user) {
        this.user = user;
    }
    //...
}
```

用户验证

4. UserDetailsService接口实现，用户读取用户信息

```
@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException(username);
        }
        return new MyUserPrincipal(user);
    }
}
```

用户验证

5. Spring配置，将自定义UserDetailsService传给DaoAuthenticationProvider，它使用用户名和密码实现认证

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
  throws Exception {
    auth.authenticationProvider(authenticationProvider());
}

@Bean
public DaoAuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider authProvider
      = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(userDetailsService);
    authProvider.setPasswordEncoder(encoder());
    return authProvider;
}

@Bean
public PasswordEncoder encoder() {
    return new BCryptPasswordEncoder(11);
}
```

对应的XML配置

```
<bean id="myUserDetailsService"
  class=“org.dharma.security.MyUserDetailsService"/>

<security:authentication-manager>
    <security:authentication-provider
      user-service-ref="myUserDetailsService" >
        <security:password-encoder ref="passwordEncoder">
        </security:password-encoder>
    </security:authentication-provider>
</security:authentication-manager>

<bean id="passwordEncoder"
  class="org.springframework.security
  .crypto.bcrypt.BCryptPasswordEncoder">
    <constructor-arg value="11"/>
</bean>
```

最多尝试次数

防止用户暴力破解密码，通过记录来自同一IP的失败次数，如果超过一定限额，阻止该账号24小时内再次登录

1. 定义登录尝试服务，用于定义最大尝试次数，记录IP，认证结果（失败，成功）

1.1 构造函数定义cache，过期时间为1天

```
public LoginAttemptService() {
    super();
    attemptsCache = CacheBuilder.newBuilder().
      expireAfterWrite(1, TimeUnit.DAYS).build(new CacheLoader<String, Integer>() {
        public Integer load(String key) {
            return 0;
        }
    });
}
```

最多尝试次数

1.2 登录成功，删除cache中该记录

```
public void loginSucceeded(String key) {
    attemptsCache.invalidate(key);
}
```

1.3 登录失败，更新cache失败次数

```
public void loginFailed(String key) {
    int attempts = 0;
    try {
        attempts = attemptsCache.get(key);
    } catch (ExecutionException e) {
        attempts = 0;
    }
    attempts++;
    attemptsCache.put(key, attempts);
}
```

1.4 读取cache判断错误次数是否超过最大尝试次数

```
public boolean isBlocked(String key) {
    try {
        return attemptsCache.get(key) >= MAX_ATTEMPT;
    } catch (ExecutionException e) {
        return false;
    }
}
```

1.5 在UserDetailsService中的loadUserByUsername方法判断IP是否被阻止

```
String ip = getClientIP();
if (loginAttemptService.isBlocked(ip)) {
    throw new RuntimeException("blocked");
}
```

最多尝试次数

2. 监听AuthenticationFailureBadCredentialsEvent事件，用于记录当前IP失败的认证

```
@Component
public class AuthenticationFailureListener
  implements ApplicationListener<AuthenticationFailureBadCredentialsEvent> {

    @Autowired
    private LoginAttemptService loginAttemptService;

    public void onApplicationEvent(AuthenticationFailureBadCredentialsEvent e) {
        WebAuthenticationDetails auth = (WebAuthenticationDetails)
          e.getAuthentication().getDetails();

        loginAttemptService.loginFailed(auth.getRemoteAddress());
    }
}
```

最多尝试次数

3. 监听AuthenticationSuccessEvent事件，用于记录当前IP成功的认证

```
@Component
public class AuthenticationSuccessEventListener
  implements ApplicationListener<AuthenticationSuccessEvent> {

    @Autowired
    private LoginAttemptService loginAttemptService;

    public void onApplicationEvent(AuthenticationSuccessEvent e) {
        WebAuthenticationDetails auth = (WebAuthenticationDetails)
          e.getAuthentication().getDetails();

        loginAttemptService.loginSucceeded(auth.getRemoteAddress());
    }
}
```

最多尝试次数

4. web.xml增加配置如下，用于在UserDetailsService服务获取request

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

5. 定义SimpleUrlAuthenticationFailureHandler错误处理，定义并抛出错误消息

```
@Component
public class CustomAuthenticationFailureHandler extends SimpleUrlAuthenticationFailureHandler {
    @Autowired
    private MessageSource messages;
    @Override
    public void onAuthenticationFailure(...) {

        String errorMessage = messages.getMessage("message.badCredentials", null, locale);
        if (exception.getMessage().equalsIgnoreCase("blocked")) {
            errorMessage = messages.getMessage("auth.message.blocked", null, locale);
        }
    }
}
```

复杂权限管理

1. 实现***PermissionEvaluator***接口，自定义安全表达式***hasPermission***

```
public class CustomPermissionEvaluator implements PermissionEvaluator {
    @Override
    public boolean hasPermission(
      Authentication auth, Object targetDomainObject, Object permission) {
        ...
        return hasPrivilege(auth, targetType, permission.toString().toUpperCase());
    }

    @Override
    public boolean hasPermission(
      Authentication auth, Serializable targetId, String targetType, Object permission) {
        ...
        return hasPrivilege(auth, targetType.toUpperCase(),
          permission.toString().toUpperCase());
    }
}
```

复杂权限管理

2. 实现*MethodSecurityExpressionOperations* 接口，定义和复写安全表达式

```
public class MySecurityExpressionRoot implements MethodSecurityExpressionOperations {
    ...
    @Override
    public final boolean hasAuthority(String authority) {
        if(authority.contains("READ")) {
            return true;
        } else {
            throw new RuntimeException("method hasAuthority() not allowed");
        }
    }

    public boolean isMember(Long OrganizationId) {
        final User user = ((MyUserPrincipal) this.getPrincipal()).getUser();
        return user.getOrganization().getId().longValue() == OrganizationId.longValue();
    }
```

复杂权限管理

```
private PermissionEvaluator permissionEvaluator;

...
@Override
public boolean hasPermission(Object target, Object permission) {
    return permissionEvaluator.hasPermission(authentication, target, permission);
}

@Override
public boolean hasPermission(Object targetId, String targetType, Object permission) {
    return permissionEvaluator.hasPermission(authentication, (Serializable) targetId, targetType, permission);
}

...
```

复杂权限管理

3. 增加方法级别安全配置，使用自定义安全表达式实现授权

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        DefaultMethodSecurityExpressionHandler expressionHandler =
          new DefaultMethodSecurityExpressionHandler();
        expressionHandler.setPermissionEvaluator(new CustomPermissionEvaluator());
        return expressionHandler;
    }
}
```

4. 控制器中调用安全表达式

```
@PostAuthorize("hasPermission(returnObject, 'read')")
```

Thanks!

Any questions?