**Spring**  là một framework , cung cấp các tính năng bảo mật khác nhau như : authentication, authorization để tạo các ựng dụng bảo mật Java Enterprise

Nó là một tiểu dự án của Spring framework đc bắt đầu vào năm 2003 bởi Ben Alex. Sau đó , vào năm 2004 , nó được phát hành theo Apache License là Spring Security 2.0.0

Nó tránh tất cả vấn đề xảy ra trong quá trình tạo các ứng về security của Spring , quản lý server environment cho ứng dụng

Framework này nhắm vào 2 phần chính của ứng dụng là :

Authentication

Aithorization

**Authentication** là quá trình biết và xác định người dùng muốn truy cập

**Authorization** là quá trình cho phép người dùng thực hiện các action trong ứng dụng

Chúng ta có thể áp dụng ủy quyền cho phép request web, method and truy cập vào từng demain riêng lẻ.

Spring Security framework hỗ trợ nhiều loại mô hình xác thực. Mô hình này được cung cấp bởi bên thứ 3 hoặc chính nó. Spring Security hỗ trợ tích hợp với tất cả các loại công nghệ này:

HTTP BASIC authentication header

HTTP Diagest authentication headers

HTTP X.509 client certificate exchange

LDAP ( Lighweight directory Access Protocol )

Form-based authentication

OpenID authentication

Automatic remember-me authentication

Kerberos

JOSSO ( Java Open Source Signle Sign-On )

AppFuse

AndroMDA

DWR ( Direct web request )

Framework này có bản chất xác thực linh hoạt đẻ tích hợp với bất kỳ giải pháp phần mềm nào . Đôi khi , các develop muốn tích hợp nó với một hệ thống kế thừa không tuân theo bất kỳ tiêu chuẩn bảo mật nào. Spring Secutiry khá độc đáo

**Advantages**

Spring Security có khá nhiều ưu điểm :

Hỗ trợ toàn diện cho authentication và authorization

Bảo vệ chống lại các tác vụ phổ biến

Tích hợp Api Servlet

Tích hợp với Spring MVC

Tính di động

CSRF protection

Java Configuration support

**Spring Security History**

Cuối năm 2003, một dự án Acegi Security System cho Spring bắt đầu với ý định phát triển một hệ thống security dựa trên Spring . Vì vậy , một hệ thông security đơn giản đã được thực hiueenj nhưng ko đc phát hành chính thức. Các developer đã sử dụng code đó trong nội bộ cho các giải pháp của họ.

Ban đầu, module authentication ko phải là 1 phần của dự án, khoảng một năm sau , module được thêm vào và dự án hoàn chỉnh được cấu hình lại để hỗ trợ nhiều công nghệ hơn.

Sau một thời gian , dự án trở thành một sub-project của Spring Framework và được phát hành 1.0.0 vào năm 2006

Trong năm 2007 , dự án đổi tên thành Spring Security và đc chấp nhận rộng rãi. Hiện tại nó được các develop công nhận và hỗ trợ mở rộng trên toàn thế giới.

**Spring Security Features**

LDAP ( Lighweight Directory Access Protocol )

Single sign-on

JAAS ( Java Authentication and Authorization Service ) LoginModule

Basic Access Authentication

Digest Access Authentication

Remember-me

Web Form Authentication

Authorition

Software Localization

HTTP Authorization

**LDAP ( Lighweight Directory Access Protocol )**

Nó là một giao thức ứng dụng mở để duy trì và truy cập các dịch vụ thông tin thư mục phân tán qua một giao thức internet

**Single sign-on**

Tính năng này cho phép người dùng truy cập nhiều ứng dụng với sự trợ giúp của một tài khoản ( username , password )

**JAAS ( Java Authentication and Authorization Service ) LoginModule**

Là module authentication Pluggable được thực hiện trong Java. Spring Security hỗ trợ nó cho quá trình xác thực của nó

**Basic Access Authentication**

Spring Security hỗ trợ Basic Access Authentication được sử dụng để cung cấp tên người dùng và mật khảu trong khi thực hiện yêu cầu qua network

**Digest Access Authentication**

Tính năng này cho phép thực hiện quy trình xác thực an toàn hơn Basic Access Authentication . Nó yêu cầu trình duyệt xác nhận danh tính của người dùng trước khi gửi dữ liệu nhạy cảm qua mạng.

**Remember-me**

Spring Security hỗ trợ tính năng này với sự trợ giúp của HTTP Cookies. Nó nhớ cho người dùng và tránh đăng nhập lại từ cùng một máy cho đến khi người dùng đăng xuất

**Web Form Authentication**

Trong quá trình này, web form collect và authentication use xác thực thông tin xác thực người dùng trình duyệt web. Spring Secutiry hỗ trợ nó khi trong khi muốn thực hiện biểu mẩu web

**Authorization**

Spring Security cung cấp tính năng này để ủy quyền cho người dùng trước khi truy cập tài nguyên. Nó cho phép các nhà phát triển xác định chính sách truy cập đối với các tài nguyên.

**Software Localization**

Tính năng này cho phép tạo user interface bằng bất kì ngôn ngữ nào.

**HTTP Authorization**

Spring cung cấp tính năng này để ủy quyền HTTP các URL yêu cầu web bằng cách sử dụng đường dẫn Apache Ant hoặc biểu thức chính quy.

**Features added in Spring Security 5.0**

**OAuth 2.0 Login**

Tính năng cung cấp cơ sở cho mọi người dùng đăng nhập vào ứng dụng bằng cách sử dụng tài khoản hiện có của họ tại Github hoặc Google.

Tính năng này được thực hiện bằng cách sử dụng Authorization Code Grant được chỉ định trong OAuth 2.0 Authorization Framework

**Reactive Support**

Từ phiên bản Spring Secutiry 5.0 , nó cung cấp khả năng lập trình phản ứng và hỗ trợ thời gian chạy react web và có thể tích hợp với Spring WebFlux

**Modernized Password Encoding**

Spring Security 5.0 giới thiệu bộ mã hóa mật khẩu mới : DelegatingPasswordEncoder hiện đại hóa hơn và giải quyết mọi vấn đề của bộ mã hóa trước NoOpPasswordEncoder

**Cơ chế hoạt động của Spring Security**

Spring security sẽ tự động tạo form đăng nhập nếu không định nghĩa cho nó, sau khi đưang nhập một đối tượng user sẽ được lưu trong session , đối tượng user này sẽ bao gồm các thong tin như username, password, role…

**Lợi ích của Spring Security**

Lợi ích lớn nhất của Spring Secutiry là giúp bạn tích hợp tính năng xác thực và phân quyền một cách dễ

dàng vào ứng dụng của mình

Ngoài ra Spring Security còn giúp chống lại CSRF attack bảo vệ session fixation mã hóa mật khẩu.

CacheControl X-XSS-Protection

**Java Configuration**

Spring Security 3.2 đã có hỗ trợ Spring Sercurity Java Configuration cho phép người dùng dễ cấu hình Spring Security mà ko cần sử dụng bất kỳ XML nào

Có điểm tương đồng giữa Spring Security và Security Java Configuration

Cấu hình có filter Servlet được gọi là SpringSecurityFilterChain ( chịu trách nhiệm cho tất cả các bảo mật, xác thực tên người dùng mật khẩu đã gửi chuyển hướng đến form đăng nhập … ) trong ứng dụng

Example cho Spring Security Java Configuration :

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.security.config.annotation.authentication.builders.\*;

**import** org.springframework.security.config.annotation.web.configuration.\*;

*@EnableWebSecurity*

**public** **class** WebSecurityConfig **implements** WebMvcConfigurer {

*@Bean*

**public** UserDetailsService userDetailsService() **throws** Exception {

InMemoryUserDetailsManager manager = **new** InMemoryUserDetailsManager();

manager.createUser(User.withDefaultPasswordEncoder().username("user").password("password").roles("USER").build());

**return** manager;

}

}

Tổng kết tính năng:

* Yêu cầu xác thực cho mọi URL trong ứng dụng
* Tạo login form
* Cho phép user với **Username** user và **Password** để phân quyền với form dựa trên xác thực
* Cho phép user logout
* Phòng trống CSRF attact
* Bảo vệ Session Fixation
* Tích hợp Security Header
  + HTTP Strict Transport Security cho bảo mật request
  + Tích hợp X-Content-Type-Options
  + Cache Control ( Có thể override sau bởi ứng dụng để cho phép lưu vào bộ nhớ cache các tài nguyên tĩnh)
  + Tích hợp X-XSS-Protection
  + Tích hợp X-Frame-Options để giúp phát hiện Clickjacking
* Tích hợp với Servlet API methods
  + HttpServletRequest#getRemoteUser()
  + HttpServletRequest.html#getUserPrincipal()
  + HttpServletRequest.html#isUserInRole(java.lang.String)
  + HttpServletRequest.html#login(java.lang.String, java.lang.String)
  + HttpServletRequest.html#logout()

**AbstractSecurityWebApplicationInitializer**

Cần đăng ký SpringSecurityFilterChain với war

Điều này có thể được thực hiện trong cấu hình Java hỗ trợ WebApplicationInitializer của Spring trong môi trường Servlet 3.0+

Spring Security cung cấp lớp cơ sở *AbstractSecurityWebApplicationInitializer* để đảm bảo SpringSecurityFilterChain được đăng ký cho bạn

Cách sử dụng tùy thuộc vào nếu sử dụng Spring hoặc nếu Spring Security là thành phần Spring duy nhất trong ứng dụng .

Hỗ trợ cho cả ứng dụng ko sử dụng Spring hoặc sử dụng Spring.

**AbstractSecurityWebApplicationInitializer without Existing Spring**

Nếu ko sử dụng Spring hoặc Spring MVC , cần phải vượt qua trong WebSecurityConfig vào lớp cha để cấu hình được chọn

**import** org.springframework.security.web.context.\*;

**public** **class** SecurityWebApplicationInitializer

**extends** AbstractSecurityWebApplicationInitializer {

**public** SecurityWebApplicationInitializer() {

**super**(WebSecurityConfig.**class**);

}

}

Với SecurityWebApplicationInitializer cho phép :

* Tự động đăng ký bộ lọc SpringSecurityFilterChain cho mọi URL trong ứng dụng.
* Thêm một ContextLoaderListener tải WebSecurityConfig

**AbstractSecurityWebApplicationInitializer with Spring MVC**

Sử dụng WebApplicationInitializer để tải cấu hình Spring. Nhưng có thể xảy ra lỗi , thay vào đó để đăng ký Spring Security với ApplicationContext .

Vì vậy nếu sử dụng Spring MVC, SecurityWebApplicationInitializer :

**import** org.springframework.security.web.context.\*;

**public** **class** SecurityWebApplicationInitializer

**extends** AbstractSecurityWebApplicationInitializer {

}

Điều này chỉ đơn giản là sẽ ddawngkys SpringSecurityFilterChain Filter cho mỗi URL trong ứng dụng. Sau đó chúng ta sẽ đảm bảo WebSecurityConfig đã được nạp trong Applicationtializer hiện có

**public** **class** MvcWebApplicationInitializer **extends**

AbstractAnnotationConfigDispatcherServletInitializer {

*@Override*

**protected** Class<?>[] getRootConfigClasses() {

**return** **new** Class[] { WebSecurityConfig.**class** };

}

*// ... other overrides ...*

}

**HttpSecurity**

WebSecurityConfig chỉ chứa thông tin về cách xác thực người dùng. WebSecurityConfigurerAdapter cung cấp cấu hình mặc định trong phương thức configure ( HttpSercurity http ) giống như sau :

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.formLogin()

.and()

.httpBasic();

}

Cấu hình mặc định ở trên :

* Đảm bảo rằng mọi y êu cầu cho ứng dụng yêu cầu người dùng phải xác thực
* Cho phép người dùng xác thực bằng đăng nhập dựa trên form login
* Cho phép người dùng xác thực bằng xác thực HTTP cơ bản

Cấu hình trên giống với cấu hình trên XML

<http>

<intercept-url pattern="/\*\*" access="authenticated"/>

<form-login />

<http-basic />

</http>

Cấu hình Java tương đương với việc đóng một thẻ XML được thể hiện bằng cách sử dụng method and() cho phép chúng ta tiếp tục cấu hình parent

Cấu hình Java có các URL và thông số mặc định khác nhau. Điều này tạo ra các trang đăng nhập tùy chỉnh. Sử dụng Spring Security ngăn ngừa rỏ rỉ thông tin.

**Java Configuration and Form Login**

Cấu hình mặc định của Spring Security không đặt URL cho trang đăng nhập 1 cách rõ ràng

Spring Security tạo một cách tự động , dựa trên các tính năng đưuọc bật và sử dụng các giá trị chuẩn bị cho URL xử lý thông tin đăng nhập được gửi, URL mục tiêu mặc định mà người dùng đưuọc gửi đến sau khi đăng nhập.

* Đăng nhập tự động

Trong khi trang login được tự động thuận tiện để thiết lập và chạy nhanh, hầu hết các ứng dụng sẽ muốn cung cấp trang đăng nhập riêng

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.formLogin()

.loginPage("/login") 1

.permitAll(); 2

}

1 – Cấu hình được cập nhật chỉ định vị trí của trang đăng nhập

2 - .permitAll() cho tất cả người dùng có thể truy cập vào trang đăng nhập . formLogin.AllowAll() cho phép quyền truy cập cho tất cả người dùng cho tất cả các URL được liên kết với Form Login

Trang đăng nhập ví dụ được triển khai với JSP cho cấu hình hiện tại :

<c:url value="/login" var="loginUrl"/>

<form action="${loginUrl}" method="post"> 1

<c:if test="${param.error != null}"> 2

<p>

Invalid username and password.

</p>

</c:if>

<c:if test="${param.logout != null}"> 3

<p>

You have been logged out.

</p>

</c:if>

<p>

<label for="username">Username</label>

<input type="text" id="username" name="username"/> 4

</p>

<p>

<label for="password">Password</label>

<input type="password" id="password" name="password"/> 5

</p>

<input type="hidden" 6

name="${\_csrf.parameterName}"

value="${\_csrf.token}"/>

<button type="submit" class="btn">Log in</button>

</form>

**Authorize Requests**

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests() 1

.antMatchers("/resources/\*\*", "/signup", "/about").permitAll() 2

.antMatchers("/admin/\*\*").hasRole("ADMIN") 3

.antMatchers("/db/\*\*").access("hasRole('ADMIN') and hasRole('DBA')") 4

.anyRequest().authenticated() 5

.and()

*// ...*

.formLogin();

}

Chỉ định các yêu cầu tùy chỉnh cho URL của chúng tôi bằng cách them nhiều child của method http.authorizeRequest()

1 - Có nhiều child cho http.authorizeRequests() method mỗi đối sánh được xem xét theo thứ tự chúng đc khai báo

2 – Chỉ định nhiều URL pattern mà bất ký người dùng nào cũng có thể truy cập nếu url bắt đầu bằng /resources/, /signup, /about

3 – Bất kỳ URL nào bắt đầu bằng /admin sẽ bị hạn chế đối với những người dùng có vai trò ROLE\_ADMIN. Chúng ta đang gọi method hasRole() mà ko cần phải chỉ định rõ tiền tố ROLE\_

4 – Bất kỳ URL nào bắt đầu /db/ đều yêu cầu người dùng phải có cả ROLE\_ADMIN và ROLE\_DBA với đang sử dụng hasRole ko cần chỉ định rõ tiền tố ROLE\_

5 – Bất kỳ URL nào chưa được khớp trên chỉ yêu cầu người dùng xác thực

**Handling Logouts**

Khi sử dụng WebSecurityConfigurerAdapter các khả năng đăng xuất sẽ tự động được áp dụng. Mặc định là việc truy cập URL/logout sẽ được xuất người dùng bằng cách :

* Vô hiệu hóa HTTP session
* Dọn dẹp mọi xác thực rememberMe đã được cấu hình
* Xóa SecurityContextHolder
* Chuyển tiếp đến /login?logout

Tương tự như cấu hình khả năng login. Việc logout có thể them tùy chỉnh khi logout .

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.logout() 1

.logoutUrl("/my/logout")  2

.logoutSuccessUrl("/my/index") 3

.logoutSuccessHandler(logoutSuccessHandler) 4

.invalidateHttpSession(true) 5

.addLogoutHandler(logoutHandler)  6

.deleteCookies(cookieNamesToClear) 7

.and()

...

}

1 – Cung cấp hỗ trợ logout. Được tự động áp dụng khi sử dụng WebSecurityConfigurationAdapter

2 – URL kích hoạt logout xảy ra ( mặc định là /logout ) . Nếu tính bảo vệ CSRF được bật ( mặc định ) , thì yêu cầu cũng phải là POST .

3 – URL để chuyển hướng khi logout thành công ( Mặc định là /login )

4 – Chỉ định việc logout bằng LogoutSucessHandler tùy chỉnh. Nếu đều này xác định , logoutSuccessUrl() sẽ bị bỏ qua.

5 – Chỉnh định có nên vô hiệu hóa HttpSession tại thời điểm đăng xuất hay ko. Mặc định sẽ là true

6 – SecurityContextLogoutHandler được them vào LogoutHandler cuối cùng theo mặc định

7 – Chỉ định tên của Cookie sẽ bị xóa khi đăng xuất thành cong . Đây là phím tắt để them 1 CookieClearingLogoutHandler một cách rõ ràng.

Lưu ý : Logout có thể cấu hình bằng cách sử dụng XML.

Nói chúng, để tùy chỉnh logout thì có thể them LogoutHandler hoặc triển khai LogoutSuccessHandler. Đối với nhiều trường hợp phỏ biến, các trình xử lý này được áp dụng dưới các API

**LogoutHandler**

Việc triển khai LogoutHandler chỉ các lớp có thể tham gia vào việc xử lý logout, Họ dự kiến sẽ được gọi để thực hiện dọn dẹp cần thiết. Như vậy ko nên ném ra ngoại lệ. Các triển khai được cung cấp:

* PersistentTokenBasedRememberMeServices
* TokenBasedRememberMeServices
* CookieClearingLogoutHandler
* CsrfLogoutHandler
* SecurityContextLogoutHandler

Thay vì triển khai các interface trực tiếp thì có cách khác là sử dụng API để triển khai tương ứng như implement interface

* Cần được tìm hiểu thêm ( Những vấn đề này chưa tìm hiểu kỹ )

**WebFlux Security**

Spring Security Webflux hỗ trợ dựa trên WebFilter và hoạt động tương tự cho Spring WebFlux và Spring WebFlux.Fn .

**Cấu hình tối thiếu bảo mật WebFlux**

*@EnableWebFluxSecurity*

**public** **class** HelloWebfluxSecurityConfig {

*@Bean*

**public** MapReactiveUserDetailsService userDetailsService() {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("user")

.roles("USER")

.build();

**return** **new** MapReactiveUserDetailsService(user);

}

}

Cấu hình này cung cấp hình thức xác thực cơ bản http, thiết lập authorization vào bất kỳ trang nào, thiết lập trang đăng nhập mặc định và đăng xuất mặc định, thiết lập tiêu đề HTTP lien quan đến bảo mật , baor vệ CSRF và hơn thế nữa.

**Cầu hình bảo mật WebFlux Explicit**

*@EnableWebFluxSecurity*

**public** **class** HelloWebfluxSecurityConfig {

*@Bean*

**public** MapReactiveUserDetailsService userDetailsService() {

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("user")

.roles("USER")

.build();

**return** **new** MapReactiveUserDetailsService(user);

}

*@Bean*

**public** SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {

http

.authorizeExchange()

.anyExchange().authenticated()

.and()

.httpBasic().and()

.formLogin();

**return** http.build();

}

}

Cấu hình này thiế lập rõ ràng tất cả những thứ giống như cấu hình tối thiểu. Từ đây có thể dễ dàng thực hiện các thay đổi với các giá trị mặc định .

**OAuth 2.0 Login**

Tính năng OAuth Login cung cấp ứng dụng khả năng cho phép người dùng đăng nhập vào ứng dụng bằng cách sử dụng tài khoản hiện có của họ tại OAuth 2.0 Provider ( GitHub ) hoặc OpenID Connect 1.0 Provider ( Google ) .

OAuth 2.0 triển khai các trường hợp sử dụng để đăng nhập vào Google hoặc GitHub

OAuth 2.0 được triển khai bằng cách sử dụng Authorization Code Grant .

**Setting the redirect URL**

Redirect URI là đường dẫn trong ứng dụng mà tác nhân người dùng của người end user được chuyển hướng trở lại sau khi họ được xác thực Google và đã cấp quyền truy cập cho OAuth Application Client

**Configure application.xml**

Để có ứng dụng OAuth Client với Google , thì cần phải cấu hình ứng dụng để sử dụng OAuth Client cho authentication flow

spring:

security:

oauth2:

client:

registration: 1

google: 2

client-id: google-client-id

client-secret: google-client-secret

* spring.security.oauth2.client.registration là tiền tố thuộc tính cơ sở cho các thuộc tính OAuth Client.
* Theo sau base property là ID cho ClientRegistration chẳng hạn như Google

Thay thế client-id và client-secret bằng thông tin đăng nhập OAuthe 2.0

Tại thời điểm đăng nhập thì OAuth Client truy xuất địa chỉ email và thông tin hồ sơ cơ bản từ điểm cuối UserInfo và thiết lập authenticated session

**ClientRegistration**

ClientRegistration là một đại diện của một client registered với OAuth 2.0 hoặc OpenID Connect 1.0 Provider

Client Registration giữ thông tin , chẳng hạn như client id, client secret , authorization grant type, redirect URI , scope, authorization URI , token URI và nhiều thông tin chi tiết khác

**public** **final** **class** ClientRegistration {

**private** String registrationId; 1

**private** String clientId; 2

**private** String clientSecret; 3

**private** ClientAuthenticationMethod clientAuthenticationMethod; 4

**private** AuthorizationGrantType authorizationGrantType; 5

**private** String redirectUriTemplate; 6

**private** Set<String> scopes; 7

**private** ProviderDetails providerDetails;

**private** String clientName; 8

**public** **class** ProviderDetails {

**private** String authorizationUri; 9

**private** String tokenUri; 10

**private** UserInfoEndpoint userInfoEndpoint;

**private** String jwkSetUri; 11

**public** **class** UserInfoEndpoint {

**private** String uri; 12

**private** String userNameAttributeName; 13

}

}

}

**Spring Boot 2.0 Property Mappings**

Bảng sau là phác thảo ánh xạ các thuộc tính OAuth Client 2.0 cho các thuộc tính Client Registration

| **Spring Boot 2.0** | **ClientRegistration** |
| --- | --- |
| spring.security.oauth2.client.registration.*[registrationId]* | registrationId |
| spring.security.oauth2.client.registration.*[registrationId]*.client-id | clientId |
| spring.security.oauth2.client.registration.*[registrationId]*.client-secret | clientSecret |
| spring.security.oauth2.client.registration.*[registrationId]*.client-authentication-method | clientAuthenticationMethod |
| spring.security.oauth2.client.registration.*[registrationId]*.authorization-grant-type | authorizationGrantType |
| spring.security.oauth2.client.registration.*[registrationId]*.redirect-uri-template | redirectUriTemplate |
| spring.security.oauth2.client.registration.*[registrationId]*.scope | scopes |
| spring.security.oauth2.client.registration.*[registrationId]*.client-name | clientName |
| spring.security.oauth2.client.provider.*[providerId]*.authorization-uri | providerDetails.authorizationUri |
| spring.security.oauth2.client.provider.*[providerId]*.token-uri | providerDetails.tokenUri |
| spring.security.oauth2.client.provider.*[providerId]*.jwk-set-uri | providerDetails.jwkSetUri |
| spring.security.oauth2.client.provider.*[providerId]*.user-info-uri | providerDetails.userInfoEndpoint.uri |
| spring.security.oauth2.client.provider.*[providerId]*.userNameAttribute | providerDetails.userInfoEndpoint.userNameAttributeName |

**ClientRegistrationRepository**

ClientRegistrationRepository đóng vai trò như 1 kho lưu trữ cho OAuth 2.0/OpenID Connect 1.0 ClientRegistration

Thông tin Client Registration được lưu trữ và thuộc sở hữu của Authorzation Server. Repository này cung cấp khả năng truy xuất một tập con của thông tin ClientRegistration , được lưu trữ với Authorization Server

Spring Boot 2.0 tự động cấu hình lien kết từng thuộc tính trong spring.security.oauth2.client.registration. [RegistrationId] để có 1 instance của ClientRegistration và sau đó soạn từng ClientRegistration instance với một ClientRegistrationRepository.

Việc thực hiện mặc định của ClientRegistrationRepository là InMemoryClientRegistrationRepository

Cấu hình tự động cũng đăng ký ClientRegistrationRepository dưới dạng @Bean trong ApplicationContext để nó có sẵn trong việc DI, nếu cần thiết bởi ứng dụng.

*@Controller*

**public** **class** MainController {

*@Autowired*

**private** ClientRegistrationRepository clientRegistrationRepository;

*@RequestMapping("/")*

**public** String index() {

ClientRegistration googleRegistration =

**this**.clientRegistrationRepository.findByRegistrationId("google");

...

**return** "index";

}

}

**CommonOAuthe2Provider**

CommonOAuth2Provider định nghĩa trước 1 tập hợp thuộc tính Client property cho 1 số nhà cung cấp nổi tiếng: Google, GitHub , FaceBook và Okta

Ví dụ như : authorization-uri , token-uri, và user-info-uri ko thay đổi thưởng xuyên với Provider nên nó có ý nghĩa để cung cấp các giá trị mặc định để giảm cáu hình cần thiết.

Cấu hình sẽ yêu cầu : client-id và client-secret

spring:

security:

oauth2:

client:

registration:

google:

client-id: google-client-id

client-secret: google-client-secret

Đối với trường hợp mà muốn chỉ định một registrationID khác chẳng hạn như đăng nhập Google thì có thể sử dụng tính năng tự động mặc định thuộc tính client bằng cách cấu hình của Provider ( Nhà cung cấp)

spring:

security:

oauth2:

client:

registration:

google-login: 1

provider: google 2

client-id: google-client-id

client-secret: google-client-secret

2- Nó sẽ tận dụng tính năng tự động mặc định các thuộc tính máy khách được đặt trong CommonOAuthe2provider.GOOGLE.getBuilder()

**Configuring Custom Provider Properties**

Có một OAuthe 2.0 Providers hỗ trợ multi-tenancy ( hợp đồng thuê ) , kết quarlaf protocol enpoint khác nhau cho mỗi hợp đồng thuê

spring:

security:

oauth2:

client:

registration:

okta:

client-id: okta-client-id

client-secret: okta-client-secret

provider:

okta: 1

authorization-uri: https://your-subdomain.oktapreview.com/oauth2/v1/authorize

token-uri: https://your-subdomain.oktapreview.com/oauth2/v1/token

user-info-uri: https://your-subdomain.oktapreview.com/oauth2/v1/userinfo

user-name-attribute: sub

jwk-set-uri: https://your-subdomain.oktapreview.com/oauth2/v1/keys

Thuộc tính cơ sở spring.security.oauthe2.client.provider.okta cho phép cấu hình tùy chỉnh các vị trí điểm cuối protocol

**Overrideing Spring Boot 2.0 Auto-configuration**

The Spring Boot 2.0 tự động cấu hình class cho hỗ trợ OAuth Client là OAuth2ClientAutoConfiguration

Nó thực hiện các tác vụ sau :

* Đăng ký một ClientRegistrationREpository @Bean bao gồm ClientRegistration từ các thuộc tính OAuth Client đc cấu hình
* Cung cấp WebSecurityConfigurarerAdapter @Configuration và cho phép OAuth 2.0 Login thông qua httpSecurity.oauth2Login()

Nếu cần ghi đè cấu hình tự động dựa trên các yêu cầu cụ thể của mình , có thể thực hiện theo các cách sau :

* Register ClientRegistrationRepository @Bean
* Cung cấp WebSecurityConfigurerAdapter
* Ghi đè tự động

Với cách Register ClientRegistrationRepository @Bean

*@Configuration*

**public** **class** OAuth2LoginConfig {

*@Bean*

**public** ClientRegistrationRepository clientRegistrationRepository() {

**return** **new** InMemoryClientRegistrationRepository(**this**.googleClientRegistration());

}

**private** ClientRegistration googleClientRegistration() {

**return** ClientRegistration.withRegistrationId("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.BASIC)

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.redirectUriTemplate("{baseUrl}/login/oauth2/code/{registrationId}")

.scope("openid", "profile", "email", "address", "phone")

.authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")

.tokenUri("https://www.googleapis.com/oauth2/v4/token")

.userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")

.userNameAttributeName(IdTokenClaimNames.SUB)

.jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")

.clientName("Google")

.build();

}

}

Với cách : Cung cấp WebSecurityConfigurerAdapter

Ví dụ sau cho thấy cách cung cấp 1 WebSecurityConfigurerAdapter với @EnalbleWebSecurity và cho phép đăng nhập OAuth 2.0 thông qua httpSecurity.oauthe2Login

*@EnableWebSecurity*

**public** **class** OAuth2LoginSecurityConfig **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.oauth2Login();

}

}

Hoặc với cách Override Auto Configuration

Ví dụ sau cho thấy cách hoàn toàn ghi đè cấu hình tự động bằng cách đăng ký ClientRegistrationRepository @Bean và cung cấp 1 WebSecurityAdapter cả 2 đều được mô tả trong 2 phần trước

*@Configuration*

**public** **class** OAuth2LoginConfig {

*@EnableWebSecurity*

**public** **static** **class** OAuth2LoginSecurityConfig **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.oauth2Login();

}

}

*@Bean*

**public** ClientRegistrationRepository clientRegistrationRepository() {

**return** **new** InMemoryClientRegistrationRepository(**this**.googleClientRegistration());

}

**private** ClientRegistration googleClientRegistration() {

**return** ClientRegistration.withRegistrationId("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.BASIC)

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.redirectUriTemplate("{baseUrl}/login/oauth2/code/{registrationId}")

.scope("openid", "profile", "email", "address", "phone")

.authorizationUri("https://accounts.google.com/o/oauth2/v2/auth")

.tokenUri("https://www.googleapis.com/oauth2/v4/token")

.userInfoUri("https://www.googleapis.com/oauth2/v3/userinfo")

.userNameAttributeName(IdTokenClaimNames.SUB)

.jwkSetUri("https://www.googleapis.com/oauth2/v3/certs")

.clientName("Google")

.build();

}

}

**Java Configuration without Spring Boot 2.0**

Nếu ko sử dụng Spring Boot 2.0 và muốn cấu định cấu hình một trong các nhà cung cấp xác định trước trong CommonOAuth2Provider ( Google ) :

*@Configuration*

**public** **class** OAuth2LoginConfig {

*@EnableWebSecurity*

**public** **static** **class** OAuth2LoginSecurityConfig **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.oauth2Login();

}

}

*@Bean*

**public** ClientRegistrationRepository clientRegistrationRepository() {

**return** **new** InMemoryClientRegistrationRepository(**this**.googleClientRegistration());

}

*@Bean*

**public** OAuth2AuthorizedClientService authorizedClientService() {

**return** **new** InMemoryOAuth2AuthorizedClientService(**this**.clientRegistrationRepository());

}

**private** ClientRegistration googleClientRegistration() {

**return** CommonOAuth2Provider.GOOGLE.getBuilder("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.build();

}

}

**OAuth2AuthorizedClient / OAuth2AuthorizedClientService**

@Auth2AuthorizedClient là một đại diện của 1 Authorized Client . Client đượ c coi là được authorized khi end-user đã cấp quyền cho khách hàng truy cập các tài nguyên bảo vệ của nó.

OAuth2AuthorizedClient phục phụ mục đích kết hợp 1 OAuth2AccessToken với một chủ sở hữu ClientRegistration và tài nguyên, là người dùng cuối chính đã được cấp phép.

Vai trò chính của OAuth2AuthorizedClientService là quản lý các cá thể OAuth2AuthoriedClient

Từ góc độ developer, nó cung cấp khả năng tra cứu một OAuth2AccessToken được lien kết với một máy khách để nó có thể sử dụng để khởi tạo một yêu cầu đến một máy chủ tài nguyên

Developer cũng có thể đăng ký 1 OAuth2AuthorizedService @Bean trong ApplicationContext ( ghi đè ) để có khả năng tra cứu một OAuth2AccessToken được liên kết với một ClientRegistration cụ thể.

*@Controller*

**public** **class** MainController {

*@Autowired*

**private** OAuth2AuthorizedClientService authorizedClientService;

*@RequestMapping("/userinfo")*

**public** String userinfo(OAuth2AuthenticationToken authentication) {

*// authentication.getAuthorizedClientRegistrationId() returns the*

*// registrationId of the Client that was authorized during the Login flow*

OAuth2AuthorizedClient authorizedClient =

**this**.authorizedClientService.loadAuthorizedClient(

authentication.getAuthorizedClientRegistrationId(),

authentication.getName());

OAuth2AccessToken accessToken = authorizedClient.getAccessToken();

...

**return** "userinfo";

}

}

**Authentication**

**Inmemory Authentication**

Ví dụ để định cấu hình nhiều người dùng.

*@Bean*

**public** UserDetailsService userDetailsService() **throws** Exception {

*// ensure the passwords are encoded properly*

UserBuilder users = User.withDefaultPasswordEncoder();

InMemoryUserDetailsManager manager = **new** InMemoryUserDetailsManager();

manager.createUser(users.username("user").password("password").roles("USER").build());

manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").build());

**return** manager;

}

**JDBC Authentication**

Có thể tìm thấy các bản cập nhật đẻ hỗ trợ xác thực dựa trên JDBC. Sử dụng DataSource để xác định nguồn dữ liệu trong ứng dụng .

Mẫu cung JDBC để sử dụng xác thực

*@Autowired*

**private** DataSource dataSource;

*@Autowired*

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception {

*// ensure the passwords are encoded properly*

UserBuilder users = User.withDefaultPasswordEncoder();

auth

.jdbcAuthentication()

.dataSource(dataSource)

.withDefaultSchema()

.withUser(users.username("user").password("password").roles("USER"))

.withUser(users.username("admin").password("password").roles("USER","ADMIN"));

}

**LDAP Authentication**

Có thể tháy các bản cập nhật xác thực dựa trên LDAP . Mẫu Idap-javaconfig cung cấp 1 ví dụ hoàn chỉnh về việc sử dụng xác thực dựa trên LDAP

*@Autowired*

**private** DataSource dataSource;

*@Autowired*

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception {

auth

.ldapAuthentication()

.userDnPatterns("uid={0},ou=people")

.groupSearchBase("ou=groups");

}

**AuthenticationPrivider**

Có thể xác định xác thực tùy chỉnh bằng cách hiển thị một AuthenticationProvider tùy chỉnh làm Bean. Ví dụ sau sẽ tùy chỉnh xác thực giả định SpringAuthenticationProvider triển khai AuthenticationProvider

*@Bean*

**public** SpringAuthenticationProvider springAuthenticationProvider() {

**return** **new** SpringAuthenticationProvider();

}

Cách sử dụng bị hạn chế.

**UserDetailsService**

Có thể xác định xác thực tùy chỉnh bằng cách bày ra một UserDetailService tùy chỉnh như một Bean. Ví dụ sau sẽ tùy chỉnh xác thực giả định rằng SpringDataUserDetailsService thực hiện UserDetailsService

*@Bean*

**public** SpringDataUserDetailsService springDataUserDetailsService() {

**return** **new** SpringDataUserDetailsService();

}

Điều này chỉ sử dụng nếu AuthenticationManagerBuilder chưa được phổ biến và không có AuthenticationProviderBean được định nghĩa.

Bạn cũng có thể tùy chỉnh cách mật khẩu mã hóa bằng cách sử dụng PasswordEncoder.

Nếu sử dụng bcrypt thì có thể them như dưới đây

*@Bean*

**public** BCryptPasswordEncoder passwordEncoder() {

**return** **new** BCryptPasswordEncoder();

}

**Multiple HttpSecurity**

Chúng ta có thể cấu hình nhiefu cá thể HttpSecurity giống như chúng ta có thể có nhiều khối <http> . Điều quan trọng là mở rộng WebSecurityCOnfigurationAdapter nhiều lần.

*@EnableWebSecurity*

**public** **class** MultiHttpSecurityConfig {

*@Bean* 1

**public** UserDetailsService userDetailsService() **throws** Exception {

*// ensure the passwords are encoded properly*

UserBuilder users = User.withDefaultPasswordEncoder();

InMemoryUserDetailsManager manager = **new** InMemoryUserDetailsManager();

manager.createUser(users.username("user").password("password").roles("USER").build());

manager.createUser(users.username("admin").password("password").roles("USER","ADMIN").build());

**return** manager;

}

*@Configuration*

*@Order(1)* 2

**public** **static** **class** ApiWebSecurityConfigurationAdapter **extends** WebSecurityConfigurerAdapter {

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.antMatcher("/api/\*\*") 3

.authorizeRequests()

.anyRequest().hasRole("ADMIN")

.and()

.httpBasic();

}

}

*@Configuration* 4

**public** **static** **class** FormLoginWebSecurityConfigurerAdapter **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.formLogin();

}

}

}

1 – Config Authentication như thường  
2 – Tạo một Instance của WebSecurityConfigurerAdapter chứa đựng @Order để chỉ định WebSecurityConfigurerAdapter nên được xem xét đầu tiên.

4 – Tạo ra thể hiện khác của WebSecurityConfigurerAdapter. Nếu URL không bắt đầu bằng /adpi/ cấu ình này sẽ được sử dụng . Cấu hình nếu sử dụng @Order sau 1 k có @Order mặc định là cuôi cùng

**Method Security**

Từ sersion 2.0 trở đi , Spring Security đã cải thiện đáng kể hỗ trọ cho việc thêm bảo mật cho các method class service

Nó cung cấp hỗ trợ @Secured anotation. Từ 3.0 cũng có thể dùng anotation trên expression based anotations. Có thể áp dụng bảo mật cho 1 bean duy nhất, sử dụng phần tử method chặn để trang trí khai báo bean, hoặc có thể đảm bảo nhiều bean trên toàn bộ lớp dịch vụ bằng cách sử dụng phím tắt kiểu aspectJ

Sử dụng EnableGlobalMethodSecurity

Có thể bật tính năng bảo mật dựa trên anotation bằng cách sử dụng @EnableGlobalMethodSecurity bằng bất kỳ thể hiện @Configuration.

Ex : @EnableGlobalMethodSecurity(securedEnabled = true)

Thêm Anotation vào 1 method ( trên 1 class hoặc interface ) sau đó sẽ giới hạn quyền truy cập vào method đó cho phù hợp . Hỗ trợ annotation của Spring Security định nghĩa 1 tập hợp thuộc tính cho method. Chúng sẽ được chuyển đến AccessDecisionManager đẻ đưa ra quyết định thực tế.

**public** **interface** BankService {

*@Secured("IS\_AUTHENTICATED\_ANONYMOUSLY")*

**public** Account readAccount(Long id);

*@Secured("IS\_AUTHENTICATED\_ANONYMOUSLY")*

**public** Account[] findAccounts();

*@Secured("ROLE\_TELLER")*

**public** Account post(Account account, **double** amount);

}

Đây là tiêu chuẩn dựa trên và cho phép áp dụng các rằng buộc dựa trên vai trò đơn giản nhưng ko có annotation gốc của Spring Security . Để sử dunjgcus pháp của biểu thức thì sẽ sử dụng :

*@EnableGlobalMethodSecurity(prePostEnabled = true)*

**public** **class** MethodSecurityConfig {

*// ...*

}

Nó sẽ tương đương

**public** **interface** BankService {

*@PreAuthorize("isAnonymous()")*

**public** Account readAccount(Long id);

*@PreAuthorize("isAnonymous()")*

**public** Account[] findAccounts();

*@PreAuthorize("hasAuthority('ROLE\_TELLER')")*

**public** Account post(Account account, **double** amount);

}

EnableReactiveMethodSecurity

Spring Security hỗ trợ bảo mật method bằng cách sử dụng Context’s Context được thiết lập bằng ReactiveSecurityContextHolder . Điều này thể hiện cách truy xuất của người dụng hiện đã đăng nhập.

Để làm được việc này, kiểu trả về của method phải là org.reactivestreams.Publisher.

Authentication authentication = **new** TestingAuthenticationToken("user", "password", "ROLE\_USER");

Mono<String> messageByUsername = ReactiveSecurityContextHolder.getContext()

.map(SecurityContext::getAuthentication)

.map(Authentication::getName)

.flatMap(**this**::findMessageByUsername)

*// In a WebFlux application the `subscriberContext` is automatically setup using `ReactorContextWebFilter`*

.subscriberContext(ReactiveSecurityContextHolder.withAuthentication(authentication));

StepVerifier.create(messageByUsername)

.expectNext("Hi user")

.verifyComplete();

Với this::findMessageByUsername định nghĩa

Mono<String> findMessageByUsername(String username) {

**return** Mono.just("Hi " + username);

}

Dưới đây là cấu hình bảo mật method tối thiểu khi sử du njg bảo mật method trong các ứng dụng Reactives

*@EnableReactiveMethodSecurity*

**public** **class** SecurityConfig {

*@Bean*

**public** MapReactiveUserDetailsService userDetailsService() {

User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();

UserDetails rob = userBuilder.username("rob").password("rob").roles("USER").build();

UserDetails admin = userBuilder.username("admin").password("admin").roles("USER","ADMIN").build();

**return** **new** MapReactiveUserDetailsService(rob, admin);

}

}

Xem xét class sau :

*@Component*

**public** **class** HelloWorldMessageService {

*@PreAuthorize("hasRole('ADMIN')")*

**public** Mono<String> findMessage() {

**return** Mono.just("Hello World!");

}

}

@PreAuthorize(“hasRole ( ‘ADMIN’)”) sẽ đảm bảo rằng findByMessage chỉ được gọi bởi người dùng có Role là ADMIN. Điều quan trọng là cần lưu lý là bất biểu thức nào trong method security chuẩn cho @EnableReactiveMethodSecurity. Tuy nhiên, tại thời điểm này , nó chỉ hỗ trợ kiểu trả về Boolean hoặc Boolean của biểu thức. Điều này có nghĩa là biểu thức ko được block.

Khi tích hợp với WebFlux Security thì React se tự động thiết lập bởi Spring Security theo người dùng đã được xác thực .

*@EnableWebFluxSecurity*

*@EnableReactiveMethodSecurity*

**public** **class** SecurityConfig {

*@Bean*

SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) **throws** Exception {

**return** http

*// Demonstrate that method security works*

*// Best practice to use both for defense in depth*

.authorizeExchange()

.anyExchange().permitAll()

.and()

.httpBasic().and()

.build();

}

*@Bean*

MapReactiveUserDetailsService userDetailsService() {

User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();

UserDetails rob = userBuilder.username("rob").password("rob").roles("USER").build();

UserDetails admin = userBuilder.username("admin").password("admin").roles("USER","ADMIN").build();

**return** **new** MapReactiveUserDetailsService(rob, admin);

}

}

**Post Precessing Configured Objects**

Cấu hình Java của Spring Security không hiển thị mọi thuộc tính của mọi đối tượng mà nó cấu hình. Điều này giúp đơn giản hóa cấu hình cho đa số người dùng. Sau đó, nếu mọi thuộc tính được bày ra, người dùng có thể sử dụng cấu hình Bean tiêu chuẩn.

Mặc dù các lý do chính đáng để ko hiển thị trực tiếp mọi thuộc tính, người dùng vẫn có thể cần them các tùy chọn cấu hình nâng cao.

Để giải quyết Spring Security giới thiệu khái niệm về ObjectPostProcessor có thể sử dụng để sửa đổi hoặc thay thế nhiều cá thể Object được tạo bởi cấu hình Java

Ví dụ nếu muốn cấu hình thuộc tính filterSecurityPublistAuthorizationSuccess trên FilterSecurityInterceptor , bạn có thể sử dụng như sau :

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.withObjectPostProcessor(**new** ObjectPostProcessor<FilterSecurityInterceptor>() {

**public** <O **extends** FilterSecurityInterceptor> O postProcess(

O fsi) {

fsi.setPublishAuthorizationSuccess(true);

**return** fsi;

}

});

}

**Custom DSLs**

Có thể cung cấp các DSL tùy chỉnh của riêng bạn trong Spring Security.

Ví dụ:

**public** **class** MyCustomDsl **extends** AbstractHttpConfigurer<MyCustomDsl, HttpSecurity> {

**private** **boolean** flag;

*@Override*

**public** **void** init(H http) **throws** Exception {

*// any method that adds another configurer*

*// must be done in the init method*

http.csrf().disable();

}

*@Override*

**public** **void** configure(H http) **throws** Exception {

ApplicationContext context = http.getSharedObject(ApplicationContext.**class**);

*// here we lookup from the ApplicationContext. You can also just create a new instance.*

MyFilter myFilter = context.getBean(MyFilter.**class**);

myFilter.setFlag(flag);

http.addFilterBefore(myFilter, UsernamePasswordAuthenticationFilter.**class**);

}

**public** MyCustomDsl flag(**boolean** value) {

**this**.flag = value;

**return** **this**;

}

**public** **static** MyCustomDsl customDsl() {

**return** **new** MyCustomDsl();

}

}

Đây thực sự là cách method như HttpSecurity.authorizeRequests được thực hiện.

*@EnableWebSecurity*

**public** **class** Config **extends** WebSecurityConfigurerAdapter {

*@Override*

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.apply(customDsl())

.flag(true)

.and()

...;

}

}

**Security namespace Configuration**

Namespace Configuration đã có sẵn từ phiên bản 2.0 ở Spring Framework. Nó cho phép bổ sung Spring Bean application Context với các phần tử XML.

Một phần tử đơn giản có thể che giấu thực tế là nhiều bean và các bước xử lý đang được them vào ngữ cảnh ứng dụng.

<security:ldap-server />

Điều này đơn giản hơn nhiều so với việc kết nối các máy chủ Apache Directory server tương đương.

Cấu hình thay thế phổ biến nhất được hỗ trợ bởi các thuộc tính trên phần tử ldap-serve và người dùng bị cô lập khỏi lo lắng về việc họ tạo ra bean nào và tên thuộc tính bean là gì .

Sử dụng XML cung cấp thông tin veef thuộc tính và các phần tử có sẵn.

Để bắt đầu sử dụng **namespace** trong application context , cần phải có spring-security-config jar trên classpath. Sau đó tất cả những gì làm là them khai báo lược đồ vào tệp ngữ cảnh ứng dụng của bạn.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:security="http://www.springframework.org/schema/security"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/security

http://www.springframework.org/schema/security/spring-security.xsd">

...

</beans>

Có thể sử dụng spring security làm namespace thay vì sử dụng bean, có nghĩa là chúng ta có thể bỏ qua tiền tố trên tất cả các phần tử namespace bảo mật, làm cho nội dung dễ đọc hơn .

**Advanced Web Feature**

Nếu ứng dụng hỗ trợ cả HTTP và HTTPS và bạn yêu cầu URL cụ thể chỉ có thể được truy cập qua HTTPS thì điều này được hỗ trợ trực tiếp bằng cách sử dụng thuộc tính require-channel trên <intercept-url>

<http>

<intercept-url pattern="/secure/\*\*" access="ROLE\_USER" requires-channel="https"/>

<intercept-url pattern="/\*\*" access="ROLE\_USER" requires-channel="any"/>

...

</http>

Nếu ứng dụng sử dụng cổng ko chuẩn cho HTTP hoặc HTTPS thì có thể chỉ định danh sách các ánh xạ port như sau .

<http>

...

<port-mappings>

<port-mapping http="9080" https="9443"/>

</port-mappings>

</http>

**Session Managerment**

Có thể định cấu hình Spring Security để phát hiện việc gửi ID session k hợp lệ và chuyển hướng tới URL thích hợp thông qua .

<http>

...

<session-management invalid-session-url="/invalidSession.htm" />

</http>

Sử dụng cơ chế này để phát hiện thời gian chờ của session, nó có thể báo cáo sai nếu người dùng đăng xuất và sau đó đăng nhập lại mà ko cần đóng trình duyệt .

Điều này do cookie ko bị xóa khi vô hiệu hóa session và sẽ gửi lại ngay cả khi người dùng đã đăng xuất.

Sử dụng logout để xóa cookie JSESSESSIONID một cách rõ ràng

<http>

<logout delete-cookies="JSESSIONID" />

</http>

Concurrent Session Control

Nếu muốn đặt các rằng buộc về khả năng đăng nhập vào ứng dụng của một người dùng duy nhất, Spring Security hỗ trợ viejc với với bổ sung đơn giản sau. Trước tiên cần phải theme listener vào web.xml

<listener>

<listener-class>

org.springframework.security.web.session.HttpSessionEventPublisher

</listener-class>

</listener>

Sau đó them vào context

<http>

...

<session-management>

<concurrency-control max-sessions="1" />

</session-management>

</http>

**OpenID Support**