Spring Security Tầng kiến trúc (v1)

Contents

Hướng dẫn này là hướng dẫn vở lòng về Spring security, cung cấp thông tin cần thiết về thiết kế và cơ bản xây dựng các khối của framework. Chúng ta làm quen rất cơ bản của App security. Tuy nhiên, trong quá trình đó chúng ta có hiểu được một vài sự nhầm lẫn đã trải qua của dev người sử dụng spring. Để làm việc đó chúng ta quan sát cách security kết nối vào web applications bằng filters ,bằng sử dụng annotations, và nhiều thứ khác. Sử dụng hướng dẫn này khi bạn cần hiểu cấp high-level cách bảo việc 1 app, cách tùy chỉnh nó hoặc nếu bạn cần học cách suy nghĩ về một secure app.

Hướng dẫn này không phải là một hướng dẫn sử dụng hoặc công thức cho các cách giải quyết mà là thể hiện các vấn để thường gặp cơ bản hữu ít cho người mới cũng như là người có kinh nghiệm. Spring boot cũng được thường xuyên được tham khảo, bởi vì nó cung cấp một vài hành động mặc định cho App security, và nó có thể hữu ít để hiểu cách điều phó phù hợp với kiến trúc tổng thể.

Chú ý: Tất cả các nguyên tắc đều áp dụng tốt cho các ứng dụng không sử dụng spring boot.

Authentication and Authorization (access control)

App Security đều có các vấn đề như sau: authentication (Bạn là ai ?) và authorization (Bạn được phép làm gì) thỉnh thoảng nhiều người nói

là "access control" thay thế cho "authorization", nó gây ra nhầm lẫn nhưng nó có thể hữu ít khi hiểu nó theo cách bởi vì "authorization" không được bao phủ ở một số nơi khác. Spring security có một tầng kiến trúc thiết kế tách biệt xác thực và ủy uyền từ chiến lược các điểm mở rộng cho cả 2.

Authentication

Chiến lược chính của authentication là AuthenticationManager interface, nó có 1 method:

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

Một AuthenticationManager có thể làm 1 trong 3 thứ sau trong method authenticate

- Trả về một authentication (bình thường là authenticated=true)nếu nó kiểm chứng ràng user input hợp lệ
- Ném ra 1 lỗi AuthenticationException nếu kiểm chứng rằng user input không hợp lệ
- Trả về một Null nếu nó không thể xác định

AuthenticationException nó là rumtime Exception. Nó thường xuyên được xử lý bởi 1 ứng dụng theo cách chung chung, dựa trên kiểu cách hoặc mục đích của ứng dụng. Trong trường hợp khác user code không thường bắt và xử lí nó. VD, một web UI có thể render ra một page nói rằng xác thực lỗi và một backend http có thể trả về một mã code 401 kèm hoặc không kèm theo www-authenticate header dựa vào context.

Về việc triển khai AuthenticationManager sử dụng phổ biến nhất là ProviderManager, nó ủy quyền cho một chuỗi các đối tượng AuthenticationProvider. Một AuthenticationProvider hơi giống AuthenticationManager, nhưng nó có thêm một phương thức để chấp thuận người gọi liệu có chấp nhận cho một kiểu authenticaton :

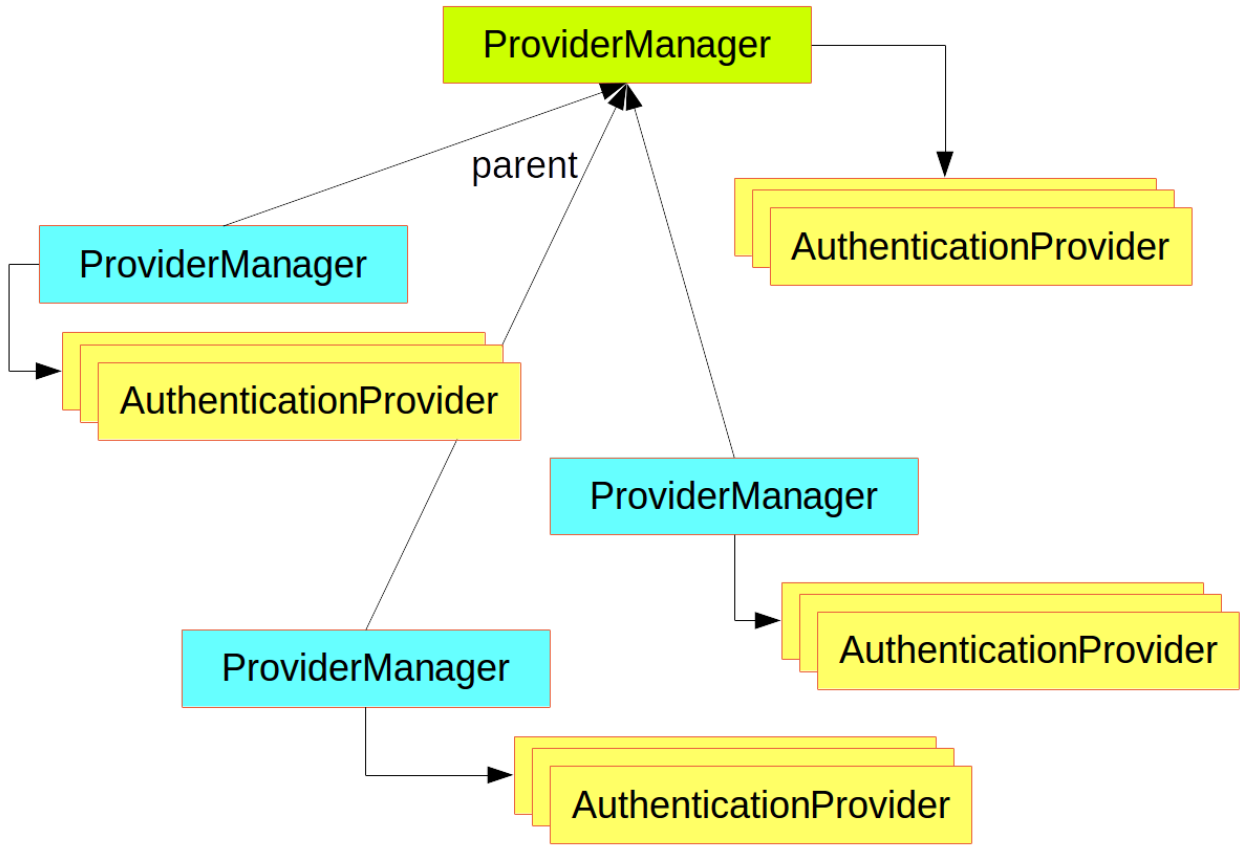
```
public interface AuthenticationProvider {
        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;
        boolean supports(Class<?> authentication);
}
```

Đối số Class<?> trong hàm supports nó thực sự là một class kế thừa Authentication (nó sẽ được gọi xem có hỗ trợ class nào đó(class para của hàm authentication()) khi tryền vào hàm authentication() hay không). Một ProviderManager có thể hỗ trợ nhiều kiểu xác thực khác nhau trong cùng 1 ứng dụng bằng các sử dụng các ủy quyền AuthenticationProvider. Nếu một ProviderManager không nhận ra một kiểu đối tượng của authentication nó sẽ bỏ qua.

Một ProviderManager có một tùy chọn tên là parent (có loại là AuthenticationManager), nó sẽ được tham chiếu nếu tất cả các AuthenticationProvider null, trường hợp nếu parent không có giá trị (Parent was null, or didn't authenticate) thì sẽ xuất ra một AuthenticationException.

Đôi khi, một ứng dụng có các logic group ứng với quyền resources (vd tất cả web resource được truy cập với các patter mẫu giống như /api/**) và mỗi goup

đó có quyền đặc trưng riêng của mình AuthenticationManager. Thông thường mỗi group logic như là một ProviderManager và chúng chia sẽ với nhau 1 parent. Parent đó được xem như là "gobal resource" thực hiện hành động như là một dự phòng cho tất cả các provider.



Hình 1: AuthenticationManager sử dụng ProviderManager

Customizing Authentication Managers

Spring security cung cấp một vài cấu hình giúp sử dụng nhanh các tính năng xác thực phổ biến trong lúc xây dựng app.
AuthenticationManagerBuidler là cách phổ biến được sử dụng, nó được sử dụng tốt để cài đặt userdetail trong in-memory, jdbc, hoặc LDAP(Lightweight Directory Access Protocol) hoặc thêm 1 custom userDetailService. Ví dụ dưới đây trình bày một app cấu hình gobal(parent) AuthenticationManager:

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
   ... // web stuff here
  @Autowired
  public void initialize(AuthenticationManagerBuilder builder, DataSource
dataSource) {
    builder.jdbcAuthentication().dataSource(dataSource).withUser("dave")
      .password("secret").roles("USER");
  }
}
```

Ở trên là ví dụ về một ứng dụng web, nhưng việc sư dụng AuthenticationMangerBuilder là được sử dụng rộng rãi. Chú ý rằng AuthenticationManagerBuilder được @Autowired trong một phương thức trong một @Bean – đó cách xây dựng một gobal(parent) AuthenticationManager. Trái ngược lại hãy xem ví dụ dưới.

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {

  @Autowired
  DataSource dataSource;
   ... // web stuff here
  @Override
  public void configure(AuthenticationManagerBuilder builder) {
    builder.jdbcAuthentication().dataSource(dataSource).withUser("dave")
      .password("secret").roles("USER");
  }
}
```

Nếu chúng ta sử dụng một @Override của một mehthod trong cấu hình, the authenticationManagerBuilder sẽ được sử dụng chỉ để build ở local AuthenticationManager, nó chỉ là con của authenticationManger toàn cầu. Trong một ứng dụng spring boot bạn có thể @Autowired the gobal AuthenticationManager vào trong 1 bean khác, nhưng bạn không thể làm đều đó với the local authenticationManager trừ khi bạn biểu thị nó một các rõ ràng.

Spring boot cung cấp một default gobal AuthenticationManager (chỉ một user) trừ phi bạn chiếm trước (pre-empt) nó bằng cách cung cấp bean kiểu AuthenticationManager của chính bạn. Mặc định thì đã đủ bảo vệ trên chính bản thân nó nên bạn không cần lo lắng quá nhiều về nó, trừ phi bạn chủ động cần một bản custom global AuthenticationManager. Nếu bạn cần làm bất kỳ cấu hình để build một authenticationManager, bạn chỉ cần làm đều đó ở local đối với các tài nguyên đang bảo vệ và không cần quan tâm về gobal default.

Authorization or access control

Một xác thực (authen) thành công, chúng ta cần chuyển đế ủy quyền (autho) và chiến lược chính ở đây là AccessDecisionManager. Ở đây có 3 cách triển khai được cung cấp bởi framework và tất cả 3 cách đều ủy nhiệm cho một chuỗi đối tượng AccessDecisionVoter. Giống như là ProviderManager ủy nhiệm cho AuthenticationProviders.

Một AccessDecisionVoter xác định một Authentication(đại diện chính) và một đối tượng đang bảo vệ, nó được biểu thị với ConfigAttributes:

```
boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);

int vote(Authentication authentication, S object,
        Collection<ConfigAttribute> attributes);
```

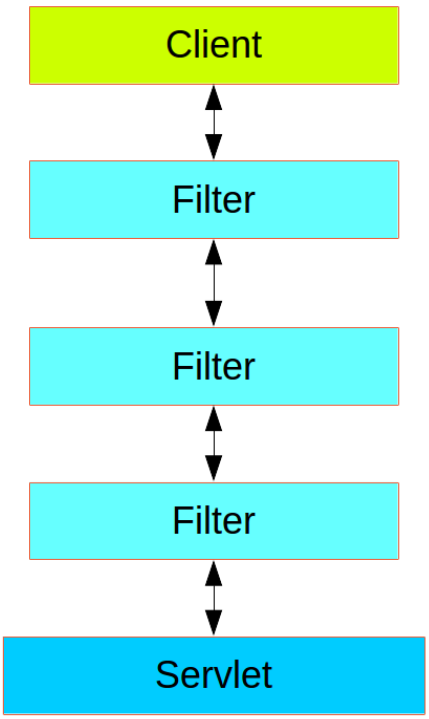
The Object đã được hoàn thành việc đăng kí với AccessDecisionManager và AccessDecisionVoter. Nó đại diện bất kể thứ gì một user muốn truy cập (một web resource hoặc một method trong một java class 2 trường hợp phổ biến). The ConfiAttibutes cũng khá chung chung, nó biểu thị cho đối tượng đang bảo vệ với một vài metadata để xác định cấp độ quyền hạn yêu cầu để truy cập Object. ConfigAttribute là một interface. Nó chỉ có một method return string, nên những string được mã hóa bằng một số cách của riêng người sở hữu resource thể hiện các quy tắc ai được chấp nhận truy cập nó. Một điển hình ConfigAttribute là name của một user role (like ROLE_ADMIN or ROLE_AUDIT) và họ thường dùng vài định dạng đăc biệt(có tiềng tố ROLE_) hoặc một đại điện biểu thức cần được đánh giá.

Hầu hết thường sử dụng default AccessDecisionManager, nó là AffirmativeBased (Nếu bất kỳ voters return khẳng định, truy cập được chấp nhận). Bất kỳ tùy chỉnh nào xảy ra trong voters bằng cách thêm những cái mới hoặc bằng cách sửa đổi những gì đang có.

Rất phổ thông để sư dụng ConfigAttribute bằng Spring Expression Language là isFullAuthenticated và hasRole('user'). Đều đó hỗ trợ bởi một AccessDecissionVoter nó có thể xử lý biểu thức và create một nội dung từ chúng. Để mở rộng biểu thức có thể xử lý yêu cầu một custom triển khai (implementation) của SecurityExpressRoot và đôi khi cũng là SecutityExpressionHandler.

Web Security

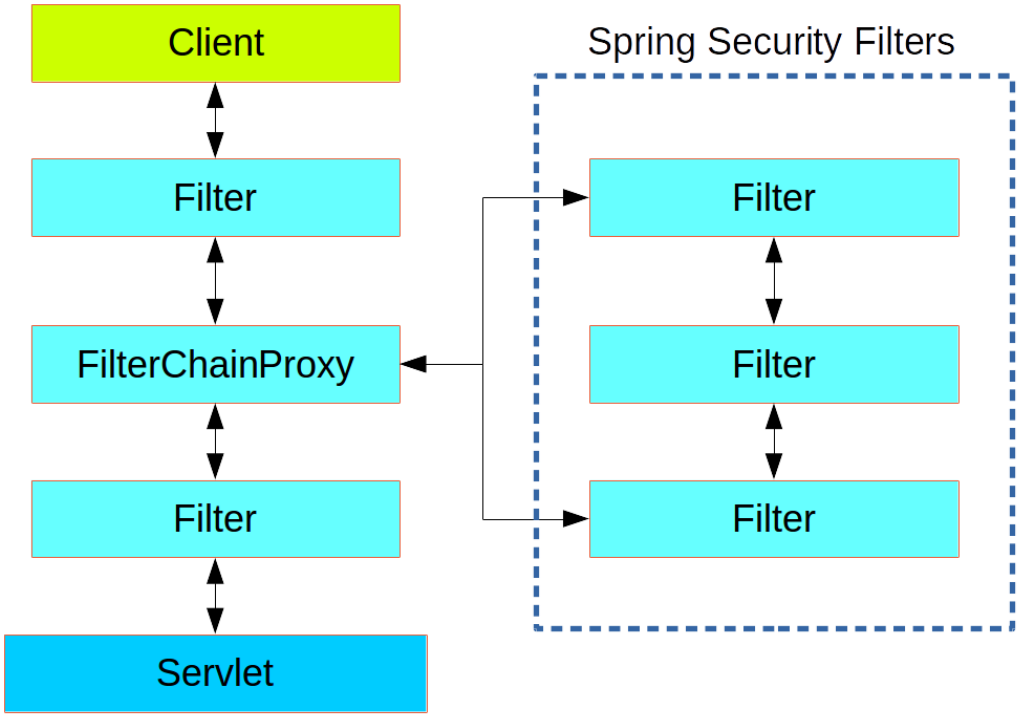
Spring security trong tầng kiến trúc web(có ui và http backends) được dựa trên servlet filters, vì vậy trước tiên sẽ hữu ích nếu xem vai trò chung của Filters. Hình ảnh bên dưới sẽ thể hiện điển hình lớp xử lý cho 1 http request.



The Client giử một request đến Application, và The container sẽ quyết định những filter nào, servlet nào tương ứng dựa trên path của URL request. Thông thường, một servlet có thể xử lý một request, nhưng những filter dạng một chuỗi nên chúng được sắp xếp. Thực tế một filter có thể phủ quyết tất cả các filter còn lại nếu nó muốn tự xử lý request tại bản thân nó. Một filter cũng có thể thay đổi request hoặc response sử dụng trong downstream filters và servlet. Vị trí sắp xếp của chuỗi filter rất quan trọng, and Spring boot quản lí nó thông qua 2 cơ chế: @Beans của kiểu Filter có thể có một @Order hoặc implement Ordered. Và chúng sẽ có thể là một phần của một FilterRegistrationBean Tự bản thân nó đã được sắp xếp. Một vài filters có sẵn đã tự định nghĩa hằng số để giúp chúng đưa ra báo hiệu thứ tự chúng muốn tương tác với nhau (Cho ví dụ, the SessionRepositoryFilter từ Spring session có Default_orther của Integer.MIN_VALUE + 50), nó sẽ báo rằng nó sẽ ở xuất hiện sớm trong chuỗi filter, nhưng nó không loại trừ các bộ lọc khác đến trước nó).

Spring security được cài đặc như là 1 single filter trong chuỗi, và cụ thể có kiểu là FilterChainProxy. Trong một spring boot application, the security filter là một @Bean trong ApplicationContext, và nó được cài đăc mặc định nên nó được ap dụng vào mỗi request. Nó sẽ được cài đặc tại vị trí được xác định bởi SpringsecurityProperties.DEFAULT_FILTER_ORDER đến FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER (vị trí cao

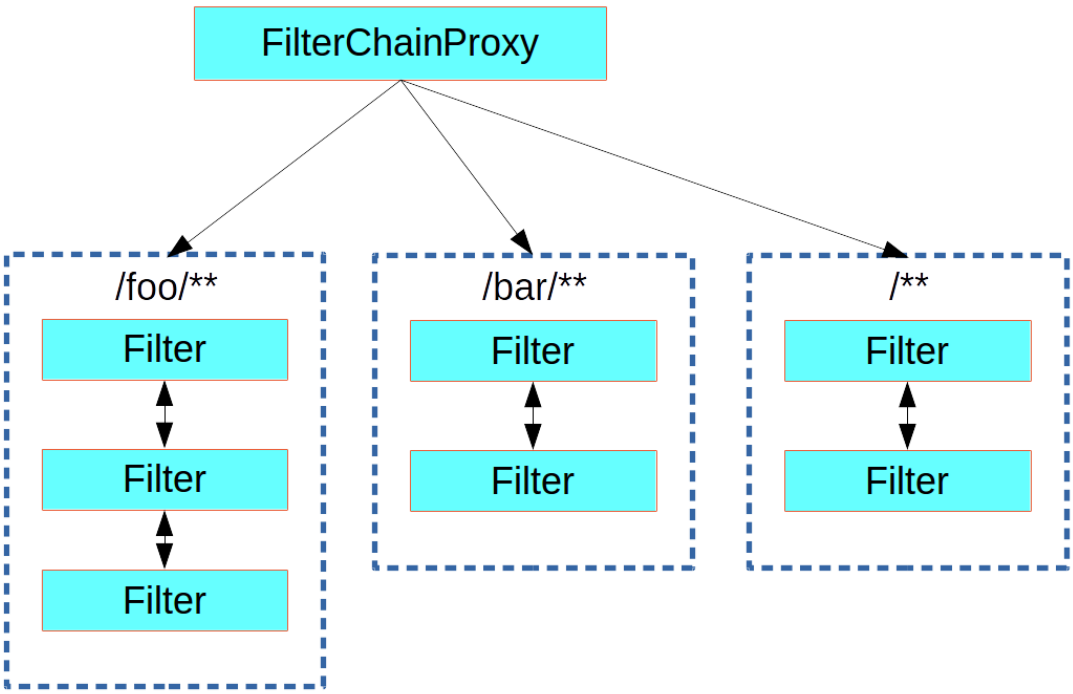
nhất mà một spring boot application có thể filter để bao bọc request, chỉnh sửa hành động). Có nhiều filter hơn ta nghĩ nếu xét góc độ của container, trong đó mỗi filter đóng 1 vai trò đặc biệt.



Hình 2 Spring security có 1 filter vật lí nhưng ủy quyền (delegate) tiến trình có một chuỗi các filter bên trong

Thực tế, còn một lớp nữa của sự chuyển hướng trong security filter: nó thường xuyên được cài đặc trong container là DelegateFilterProxy, nó không cần phải là một Spring @Bean. The proxy ủy quyền cho một FilterChainProxy, nó luôn luôn là một @Bean và luôn luôn có một name cố định là springSecutiryFilterChain. Nó là the FilterChainProxy nó chứa tất cả security logic được sắp xếp bên trong như là một chuỗi hoặc nhiều chuỗi của các filter. Tất cả các filter đều có chung API (tất cả chúng đều implement Filter interface từ servlet), và tất cả chúng đều có cơ hội phủ quyết phần còn lại của chuỗi.

Ở đó có thể có multi chuỗi filter tất cả đều được quản lí bởi Spring Security trong cùng một cấp cao nhất FilterChainProxy và tất cả đều chưa biết tới container. The spring security filter chứa một list của nhiều chuỗi filter và gửi một request đến chuỗi đầu tiên mà match với nó. Hình bên dưới thể hiện sự chuyển tiếp sảy ra dựa trên matching the request có đường dẫn /foo/** match trước /**. Đây là hết sức phổ biến nhưng nó không chỉ là là một cách để match một request. Quan trọng nhất là tính năng của tiến trình chuyển gửi (dispatch) rằng chỉ có một chuỗi xử lí một request.



Hình 3: The spring security FilterChainProxy chuyển gửi các request đến chuỗi mà match nó trước tiên.

Một vanilla Spring boot application không có sự tùy chỉnh security có một vài (gọi nó là n) chuỗi filter, ở đó thường là n=6, chuỗi đầu tiên (n-1) ở đó chỉ để bỏ qua các static resouce patterns giống như /css/** và /images/**, và error view: /error. (the path có thể được điều chỉnh bởi user với security.ignored từ cấu hình spring securityProperties). Chuỗi cuối cùng nắm bắt tất cả các path /** và nhiều hành động bao gồm xác thực, phân quyền, xử lý các exception, xử lý session, header writting, và nhiều nữa. Ở đó có tổng cộng 11 filters trong cái chuỗi mặc định này, nhưng thông thường nó không cần thiết để user quan tâm đến việc sử dụng cái nào và khi nào xài.

Chú ý: Thực tế tất cả các filters bên trong Spring Security đều không xác định bên trong container như là đều quan trọng (không cần quan tâm các filters bên trong container) đặc biệt bên trong spring boot application, ở đó mặc định tất cả @Bean của kiểu filter được đăng kí tự động with container. Nên nếu bạn muốn thêm một custom filter vào chuỗi security bạn cần đánh đấu nó không là một @Bean hoặc bọc nó trong một FilterRegistrationBean vô hiệu hóa rõ ràng the container registration.

Creating and Customizing Filter Chains

Mặc dịnh chuỗi filter dự phòng trong một spring boot application (chuỗi mà match với request /**) đã được định nghĩa sẵn thứ tự SecurityProperties.BASIC_AUTH_ORDER. Bạn có thể chuyển tắt nó bằng cách setting security.basic.enabled=false hoặc bạn có thể use nó như là một dự phòng và định nghĩa các chuỗi filter khác có rules thứ tự thấp hơp (thứ thấp hơn ưu tiên cao hơn). Để làm được đều đó thêm một @Bean của kiểu WebSecurityConfigurerAdapter (hoặc WebSecurityConfigurer) và diễn tả class với @Order.

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
      ...;
  }
}
```

Bean này sẽ chỉ spring security để thêm một chỗi filter mới và thứ tự filter của nó sẽ trước dự phòng.

Nhiều application có các quy tắc truy cập hoàn toàn khác nhau cho một bộ resource này khác với các bộ resource khác. Cho ví dụ, một application là một hosts một UI và một Backing API có thể hỗ trợ cooke-based xác thực với sự chuyển hướng đến login page nếu là phần UI truy cập và token-based xác thực với mã lỗi 401 trả về với lỗi xác thực cho request ở phần API truy cập. Mỗi tập hợp tài nguyên có WebSecurityConfigurerAdapter riêng với thứ tự duy nhất và trình đối chiếu (matcher) riêng. Nếu phần đối chiếu trùng lấp thì cái nào có thứ tự sớm nhất sẽ là chuỗi filter chiến thắng.

Request Maching for Dispatch and Authorization

Một chuỗi filter security (hoặc một webSecurityConfigurerAdapter) có một request kết nối nó sẽ được sử dụng để quyết định liệu có chấp nhật nó vào một http request hay không. Một khi đã quyết chấp nhận 1 phần của chuỗi filter thì các phần còn lại không dc chấp nhận. Tuy nhiên, bên trong 1 chuỗi filter bạn có thể có nhiều fine-grained của authorization bằng cách thêm các `antMatchers` bên trong httpSecurity .

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
      .authorizeRequests()
```

```
        .antMatchers("/match1/user").hasRole("USER")
        .antMatchers("/match1/spam").hasRole("SPAM")
        .anyRequest().isAuthenticated();
  }
}
```

Một trong những sai lầm dễ mắc phải nhất khi config spring security quên rằng những matcher này áp dụng cho các quy trình khác nhau. Một là một request khớp với toàn bộ chuỗi filter, và cái khác chỉ chọn quy tắc truy cập để áp dụng.

Combining Application Security Rules with Actuator Rules

Nếu bạn sử dụng the Spring boot Actuator cho việc quản lí endpoints, bạn có thể muốn chúng được bảo vệ, và theo mặc định chúng là như vậy. Thực tế, Bạn thêm the Actuator vào một application secure, bạn sẽ nhận thêm một filter chain nó chỉ áp dụng vào actuator endpoints. Nó được xác định với một request khớp với các matchers chỉ trên actuator endpoints và nó sẽ có 1 thứ tự là ManagementServerProperties.BASIC_AUTH_ORDER, nó ít hơn 5 lần so với filter dự phòng của default Securityproperties nên nó sẽ được tham chiếu (filter) trước filter dự phòng.

Nếu bạn muốn application của bạn chấp nhận các nguyên tắc bảo vệ vào actuator endpoints. Bạn có thể add một filter chain có thứ tự sớm hơn actuator và cái đó sẽ có request matcher bao gồm tất cả các actuator endpoints. Nếu bạn thích setting security default cho actuator endpoints, cách đơn giản nhất là thêm 1 filter chain của riêng bạn vào sau autuator nhưng sớm hơn dự phòng (ví dụ: ManagementServerProperties.BASIC_AUTH_ORDER + 1)

```
@Configuration
@Order(ManagementServerProperties.BASIC_AUTH_ORDER + 1)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/foo/**")
      ...;
  }
}
```

Chú ý: Spring security trong tầng kiến trúc web hiện tại nó ràng buộc với Servlet API, nên nó chỉ thực sự áp dụng khi run một ứng dụng trong servlet contain được nhúng hoặc bằng các cách khác. Tuy nhiên, nó không bị ràng buộc với Spring MVC hoặc phần còn lại của Stack Spring Web, vì vậy nó có thể được sử dụng trong bất kỳ ứng dụng servlet nào, vd ứng dụng sử dụng JAX-RS.

Method Security

Cũng như hỗ trợ cho việc bảo vệ ứng dụng web, Spring security cũng hỗ trợ chấp nhận các quy tắc truy cập vào method java. Đối với Spring security, đây chỉ là một "protected resource" khác. Đối với người dùng nó có nghĩa nguyên tắc truy cập được khởi tạo bằng sử dụng format của ConfigAttribute strings nhưng trên các chỗ khác trong code. Bước đầu tiên để enabled method security đối với ví dụ cấu hình cao nhất của app bạn

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {
}
```

Sau đó chúng ta có thể biểu diễn trực tiếp method

```
@Service
public class MyService {

  @Secured("ROLE_USER")
  public String secure() {
    return "Hello Security";
  }

}
```

Đây là ví dụ của một service với bảo vệ method. Nếu Spring tạo một @Bean của kiểu này, nó sẽ được ủng hộ và người gọi method sẽ phải thông qua một security interceptor trước khi thực sự thực thi method. Nếu truy cập bị từ chối, người gọi sẽ nhận được một AccessDeniedException thay vì kết quả thực sự của method.

Ở đó có các Annotaions rằng bạn có thể sử dụng trên methods để thi hành ràng buộc security là @PreAuthorize và @PostAuthorize, Nó sẽ cho phép bạn viết biểu thức chứa các tham chiếu đến các tham số của method và giá trị trả về tương ứng.

Chú ý: nó không có phổ biến cho việc phối hợp Web security và Method Security. The filte chain cung cấp tính năng trải nghiệm người dùng giống như xác thực và chuyển hướng đến login pages và vv.., và the methods security cung cấp cho việc bảo việc chi tiết hơn.

Working with Threads

Spring Security về cơ bản là ràng buộc luồn(thread-bound), bởi vì nó cần làm cho authentication chính của hiện tại được xác thực có sẵn để cho nhiều downstream consumers khác dùng. Building block căn bản là SecurityContext, nó có thể chứa một authentication (và khi một user đã logged, nó là một Authentication và chính xác nó là một authenticated). Bạn có thể luôn luôn truy cập và tính toán the SecurityContext thông qua static convenience methods trong

SecurityContextHolder đến lượt nó tính toán một ThreadLocal. Ví dụ dưới thể hiện sự sắp xếp.

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
assert(authentication.isAuthenticated);
```

Nó **không** phổ biến cho người dùng code làm đều đó, nhưng nó có thể hữu dụng nếu bạn cần viết một custom authentication filter (mặc dù sau đó ở đó có dựa trên các class trong Spring security nhưng bạn cũng nên tránh sư dụng SecurityContextHolder)

Nếu bạn cần truy cập authenticated user hiện tại trong web endpoint, bạn có thể use a method parameter trong một @RequestMapping

```
@RequestMapping("/foo")
public String foo(@AuthenticationPrincipal User user) {
  ... // do stuff with user
}
```

Annotation này kéo authentication hiện tại ra khỏi SecurityConext và gọi the getPrincipal() method trên trường parameter method. Kiểu của Principal trong một Authentication dựa trên AuthenticationManager đã từng xác nhận xác thực, vì vậy đây là cách hữu dụng để get type-safe từ data của bạn.

Nếu Spring security được sử dụng, the Pricipal từ HttpServletRequest cũng là kiểu Authenticaion vì vậy bạn cũng có thể sử dụng trực tiếp

```
@RequestMapping("/foo")
public String foo(Principal principal) {
  Authentication authentication = (Authentication) principal;
  User = (User) authentication.getPrincipal();
  ... // do stuff with user
}
```

Đều này đôi khi được hữu dụng nếu bạn cần viết code để nó hoạt động khi spring security không được sử dụng (bạn có thể cần nhiều dự phòng để load the authentication class)

5 Processing Secure Methods Asychronouly

Vì the SecurityContext là ràng buộc luồn, nếu bạn muốn làm bất ký gì với tiến trình nền (backgroud proccessing) thì phải gọi hàm bảo vệ (vd with @Sync), Bạn phải chắc rằng The context được nhân giống (propagated). Đều này tóm tắt để bao bọc SecurityContext với các task (Runnable, Callale, và vv) được thực thi trên chế độ nền. Spring security cung cấp một vài hỗ trợ để làm cho dễ dàng hơn giống như việc bao bọc Runnable và Callable. Để nhân giống được SecurityContext to @Sync mehthods, bạn cần cung cấp một AsyncConfigurer và chắc chắn ràng the Executor đúng loại.

```
@Configuration
public class ApplicationConfiguration extends AsyncConfigurerSupport {
```

```
  @Override
  public Executor getAsyncExecutor() {
    return new
DelegatingSecurityContextExecutorService(Executors.newFixedThreadPool(5));
  }

}
```