**Spring Sercurity**

Những thông tin bên dưới là của Spring Security 5.7.7-Snapshot.

# Điều kiện tiên quyết

Spring Sercurity yêu cầu java 8 trở lên

Spring Security hướng tới hoạt động khép kín, ta không cần cài đặt bất kỳ tệp cấu hình đặc biệt nào cho môi trường chạy Java đặc biệt của mình.

Đặc biệt ta không cần phải định cấu hình tệp chính sách Dịch vụ xác thực và cấp giấy phép Java (JAAS) hoặc đặt Spring Sercurity vào các vị trí classpath phổ biến.

# Có gì mới trong Spring Sercurity 5.7

## Servlet

**Web**

Được giới thiệu RequestAttributeSercurityContextRepository

Được giới thiệu SercurityContextHolderFilter – Khả năng yêu câu lưu rõ ràng SercurityContext

Đã thêm DDL hỗ trợ cho Cross Origin Policies Headers

**Ứng dụng khách Oauth 2.0**

Cho phép định cấu hình PKCE cho confidentical clients

Cho phép định cấu hình JWT assertion resolver in JwtBearerOAuth2AuthorizedClientProvider

Cho phép tùy chỉnh xác nhận quyền sở hữu trên máy khách JWT

**SAML 2.0**

Đã thêm hỗ trợ SAML 2.0 Login và Single Logout XML

## WebFlux

**Web**

Cho phép tùy chỉnh charset trong ServerHttpBasicAuthenticationConverter

Đã thêm DDL hỗ trợ cho Cross Origin Policies Headers

**Ứng dụng khách Oauth 2.0**

Cho phép định cấu hình PKCE cho confidentical clients

Cho phép định cấu hình JWT assertion resolver in JwtBearerOAuth2AuthorizedClientProvider

# Cách cài đặt Spring Sercurity

Dựa vào Maven:

<dependencies>

*<!-- ... other dependency elements ... -->*

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

</dependencies>

# Đặc trưng

## Xác thực

Spring Sercurity cung cấp hỗ trợ toàn diện cho việc xác thực.

Xác thực là cách ta xác minh danh tính của người đang cố gắng truy cập vào 1 tài nguyên cụ thể. Một cách phổ biến để xác thực người dùng là nhập tên người dùng và mật khẩu. Sau khi xác thực được thực hiện, ta biết danh tính và có thể thực hiện ủy quyền

Spring Sercurity cung cấp hỗ trợ tích hợp để xác thực người dùng. Phần này áp dụng cho cả Servlet và WebFlux.

**Lưu mật khẩu**

Interface **PasswordEncoder** được sử dụng để thực hiện **chuyển đổi 1 chiều**  của mật khẩu để cho phép mật khẩu được lữu trữ an toàn.

Đưa ra PasswordEncoder là chuyển đổi một chiều, nó không được cài đặt để chuyển đổi mật khẩu 2 chiều (tức là lữu trữ thông tin đăng nhập được sử dụng để xác thực với cơ sở dữ liệu)

Thường **PasswordEncoder** được sử dụng để **lưu trữ mật khẩu**, rồi so sánh với mật khẩu mà người dùng cung cấp tại thời điểm **xác thực** (đăng nhập, …)

**Lịch sử lưu trữ mật khẩu**

Trong suốt nhiều năm, cơ chế tiêu chuẩn để **lữu trữ mật khẩu** đã phát triển.

Ban đầu, mật khẩu được lưu trữ dưới dạng văn bản thần túy, được cho an toàn phụ thuộc vào nơi lưu trữ dữ liệu.

Tuy nhiên đã có những cách để có thể truy cập vào “kho chứa mật khẩu” đó để đánh cắp thông tin. Ví dụ: Sql Injection

Các nhà phát triền sau đó được khuyến khích lưu trữ mật khẩu được mã hóa bằng SHA-256.

Nhưng cách “thiên tài” đã nghĩ ra cách đoán các mật khẩu này, bằng cách thay vì đoán từng mật khẩu mỗi lần, những kẻ đó thực tạo ra 1 kho lưu trữ các mật khẩu có sẵn 🡪 rồi tiền hành so khớp từng mật khẩu một. Ví dụ: Rainbow tables

Để giảm thiểu thiệt hại cảu Rainbow Tables, các nhà phát triển được khuyến khích sử dụng mật khẩu **salted**. Thay vì sử dụng mật khẩu làm đầu vào cho hàm băm, các byte ngẫu nhiên được gọi là **salted** sẽ được tạo cho mọi mật khẩu của người dùng. Salted và mật khẩu của người dùng sẽ được chạy qua hàm băm để tạo ra một hàm băm duy nhất. Salted sẽ được lưu trữ cùng với mật khẩu của người dùng dưới dạng văn bản rõ ràng.

Trong thời hiện tại SHA-256 không còn an toàn nữa, do phần cứng máy tính có thể thực hiện hàng tỷ phép tính 🡪 có thể dễ dàng bẻ khóa từng mật khẩu một cách dễ dàng.

Do vậy, các nhà phát triển sau đó được khuyến khích tận dụng chức năng một chiều “thích ứng” để lưu trữ mật khẩu. Việc xác thực mật khẩu với chức năng lưu trữ một chiều “thích ứng” là việc cố ý sử dụng nhiều tài nguyên (CPU, bộ nhớ, …).

Chức năng một chiều thích ứng cho phép định cấu hình “yếu tố công việc”, có thể phát triển khi phần cứng trở lên tốt hơn

Ta lên điều chỉnh “yếu tố công việc” để mất khoảng 1 giây để xác minh mật khẩu trên hệ thống.

Sự đánh đổi này khiến cho những kẻ tấn công khó bẻ khóa mật khẩu, mà không tốn quá nhiều tài nguyên.

Spring Sercurity đã cố gắng cung cấp 1 điểm khởi đầu tốt cho “hệ số công việc”, nhưng người dùng được khuyến khích tự tùy chỉnh, do hiệu suất sẽ thay đổi đáng kể giữa các hệ thống **bcrypt**, **PBKDF2**, **scrypt**, **argon2.**

Bởi vì các chức năng một chiều thích ứng cố ý sử dụng nhiều tài nguyên, việc xác thực tên người dùng và mật khẩu cho mọi request sẽ làm giảm hiệu suất của ứng dụng một cách đáng kể. Mà không cách nào có thể làm tăng tốc độ xác thực mật khẩu vì bảo mật được tăng cường bằng cách làm cho tài nguyên xác thực trở lên chuyên sâu. **Người dùng** được khuyến khích trao đổi thông tin xác thực dài hạn (tức là tên người dùng và mật khẩu) **để lấy** thông tin xác thực ngắn hạn (như session, auth2,…). Thông tin đăng nhập ngắn hạn có thể được xác thực một cách nhanh chóng mà mất bất kỳ tổn thất nào về bảo mật.

**DelegatePasswordEncoder**

Trước Spring Sercurity 5.0, mặc định PasswordEncoder là NoOpPasswordEncoder yêu câu mật khẩu văn bản thuần tùy. Dựa trên phần **Lịch sử mật khẩu**, bạn có thể mong đợi rằng mặc định PasswordEncoder bây giờ là BcryptPasswordEncoder.

Tuy nhiên điều này bỏ qua 3 vần đề trong thế giới thực:

* Có nhiều ứng dụng sử dụng mã hóa mật khẩu cũ không thể dễ dàng di chuyển.
* Phương pháp lưu trữ mật khẩu tốt nhất sẽ lại thay đổi
* Là một khuôn khổ Spring Sercurity, do đó không thể cập nhật chính sách thường xuyên.

Thay vào đó, Spring Sercurity giới thiệu DelegatingPasswordEncodr cách giải quyết tất cả các vấn đề bằng cách:

* Đảm bảo rằng mật khảu đucợ mã hóa bằng cách sử dụng các đề xuất lưu trữ mật khẩu hiện tại.
* Cho phép xác thực mật khẩu ở các định dạng hiện tại và kế thừa.
* Cho phép nâng cấp mã hóa trong tương lại.

Ta có thể dễ dàng xây dựng 1 phiên bản DelegatingPasswordEncoder sử dụng PasswordEncoderFactories

PasswordEncoder passwordEncoder =

PasswordEncoderFactories.createDelegatingPasswordEncoder().

Ngoài ra còn có thể tạo phiên phản tùy chỉnh của riêng mình:

String idForEncode = “bcrypt”

Map encoders = new HashMap<>()

encoders.put(idForEncode, new BcryptPasswordEncoder)

encoders.put(“noop”, NoOpPasswordEncoder.getInstance)

…

PasswordEncoder passwordEncoder =

New DelegatingPasswordEncoder(idForEncode, encoders)

**Định dạng lưu trữ mật khẩu.**

Định dạng chung cho mật khẩu:

Ví dụ: Định dạng lưu trữ DelegatePasswordEncoder

{id}encodedPassword

* id là mã định danh được sử dụng để tra cứu PasswordEncoder (theo DelegatePasswordEncoder, nó là idForEncode, để tra cứu kiểu mã hóa)
* encodedPassword là mật khẩu sau khi mã hóa.

Không cần phải lo ngại, việc lưu mật khẩu có chứa id là kiểu mã hóa. Vì về cơ bản các thuật toán mã hóa đều công khai và dễ dạng tìm ra các kiểu mã hóa tương ứng với từng loại định dạng. Ví dụ: Mật khẩu mã hóa bởi Bcrypt thường bắt đầu bằng $2a$

**So khớp mật khẩu.**

Việc so khớp, trước tiên là xác định kiểu mã hóa của hệ thống.

Sau đó, là tìm danh sách mật khẩu dựa theo {id} – kiểu mã hóa và key (ví dụ: username, …)

Cuối cùng là so khớp giữa mật khẩu người dùng và mật khẩu trong hệ thống.

**Bắt đầu trải nghiệm**

Ví dụ: Đơn giản để khởi tạo User:

DelegatingPasswordEncoder.setDefaultPasswordEncoderForMatches(PasswordEncoder)

UserBuilder users = User.withDefaultPasswordEncoder();

User user = users

.username(“user”)

.password(“password”)

.roles(“USER”).build()

User admin = users

.username(“admin”)

Password(“password”)

Thao tác này thực hiện băm mật khẩu lưu trữ, nhưng mật khẩu vẫn được hiển thị trong bộ nhớ và trong mã nguồn biên dịch. Do đó được coi là không an toàn. Để đảm bảo an toàn, ta lên **băm mật khẩu ở bên ngoài** rồi nhập vào mã nguồn biên dịch. Ví dụ: sử dụng Spring boot Cli.

**Mã hóa với Spring Boot Cli**

Tìm hiểu sau.

**BcryptPasswordEncoder**

BcryptPasswordEncoder sử dụng thuật toán bcrypt để băm mật khẩu.

Để tăng khả năng chống bẻ khóa mật khẩu, bcrypt cố tình làm chậm.

Giống như các chức năng một chiều thích ứng khác, quá trình điều chỉnh sẽ mất khoảng 1 giây để xác minh mật khẩu trên hệ thống 🡪 người dùng được khuyến nghị nên tự điều chình thời gian

Ví dụ:

BcryptPasswordEncoder encoder = new BcryptPasswordEncoder(16)

String result = encoder.encode(“myPassword”)

assertTrue(encoder.matches(“myPassword”, result))

**Argon2PasswordEncoder**

Tương tự BcryptPasswordEncoder, nhưng đòi hỏi lượng bộ nhớ lớn.

**Pbkdf2PasswordEncoder**

Tương tự BcryptPasswordEncoder. Đây là một lựa chọn tốt khi cần chứng chỉ **FIPS**.

**ScryptPasswordEncoder**

Tương tự Argon2PasswordEncoder.

**Cấu hình lưu trữ mật khẩu**

Spring Sercurity sử dụng DelegatePasswordEncoder theo mặc định. Tuy nhiên, điều này có thể được tùy chỉnh bằng cách hiển thị một PasswordEncoder như một Spring Bean.

**Thay đổi cấu hình mật khẩu**

Hầu hết, ứng dung cho phép người dùng lưu mật khẩu, cũng cung cấp tính năng cập nhật mật khẩu.

[A Well-Known URL for changing Password](https://translate.google.com/website?sl=auto&tl=vi&hl=vi&client=webapp&u=https://w3c.github.io/webappsec-change-password-url/) có cở chế mà người quản lý mật khẩu có thể phát hiện ra lần cuối cập nhật mật khẩu của một ứng dụng nhất định.

Ta có thể cấu hình Spring Security để cung cấp điểm cuối này.

Ví dụ: nếu điểm cuối thay đổi mật khẩu trong ứng dụng là /change**-**password **,** thì ta có thể định cấu hình Spring Sercurity như sau:

http.passwordManagement(Customizer.withDefaults())

Sau đó, khi trình quản lý điều hướng đến /.well-know/change-password, Spring Security sẽ chuyển hướng đến điểm cuối của ta /change-password.

Hoặc, nếu điểm cuối của ta là một cái gì đó khác /change-password, ta cũng có thể chỉ định như sau:

http.passwordManagement((management) 🡪 management

.changePasswordPage(“/update-password”)

)

Với cấu hình trên, khi một trình quản lý mật khẩu điều hướng đến /.well-know/change-password, thì Spring Security sẽ điều hướng đến /update-password

## Bảo vệ chống lại sự lợi dụng

* CSRF
* Tiêu đề HTTP
* Yêu cầu HTTP

## Yêu cầu trên nhiều trang web giả mạo – CSRF

Tấn công CSRF là gì?

Ví dụ:

Giả sử rằng trang Web ngân hàn của ta cung cấp 1 biểu mẫu cho phép chuyển tiền từ người dùng hiện đang đăng nhập sang một tài khoản ngân hàng khác. Ví dụ biểu mẫu chuyển tiền giống như sau:

<form method=”post” action=”/transfer”>

<input type=”text” name=”amount”>

<input type=”text” name=”routingNumber”>

<input type=”text” name=”account”>

<input type=”submit” value=”Transfer”>

</form>

Http request tương ứng giống như sau:

POST /transfer HTTP/1.1

Host: bank.example.com

Cookie: JSESSION=randomid

Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876

Bây giờ, giả sử ta xác thực vào trang web của ngân hàn và nhưng không đăng xuất. Tiếp đến truy cập vào một trang web xấu, có mã độc hại như sau:

<form method=”post” action=”https://bank.example.com/transfer”>

<input type=”hidden” name=”amount” value=”100.00”>

<input type=”hidden” name=”routingNumber” value=”evilsRoutingNum”>

<input type=”hidden” name=”account” value=”evils”>

<input type=”submit” name=”Win money”>

</form>

Ta bấm “Win money”, trong quá trình này ta đã gửi 100 đô cho kẻ xấu. Bởi vì, tuy trang web xấu đó không thể nhìn thấy cookie của ta, nhưng các cookie của ngân hàng đã được gửi cùng với request.

Bảo vệ chống lại các cuộc tấn công CSRF

Lý do có thể xảy ra tấn công CSRF là request http từ trang web nạn nhân và request từ trang web của kẻ tấn công hoàn toàn giống nhau. Điều này có nghĩa là không có cách nào để từ chối các yêu cầu đến từ các trang web xấu.

Để bảo vệ khỏi các cuộc tấn công CSRF, ta cần đảm bảo có điều gì đó trong request mà trang web xấu không thể cung cấp để ta có thể phân biệt giữa 2 request.

Spring cung cấp 2 cơ chế để bảo vệ chống lại các cuộc tấn công CSRF:

* Synchronizer token pattern
* Chỉ định thuộc tính SameSite trong session cookie

Cả hai đều yêu cầu Safe Method must be Idempotent

Phương pháp an toàn phải lý tưởng.

Để bảo vệ chống lại CSRF hoạt động, ứng dụng phải đảm bảo rằng các phương thức HTTP “an toàn” phải lý tưởng.

Điều này có nghĩa là các request HTTP: GET, POST, OPTIONS, TRACE không được thay đổi HEAD.

Synchronizer token pattern

Cách chủ yếu và toàn điện nhất để chống lại các cuộc tấn công CSRF là sử dụng synchronnizer token pattern.

Khi một request http được gửi, máy chủ phải tra cứu token CSRF cung cấp và so sánh với token CRSF hệ thống. Nếu các giá trị không khớp, HTTP sẽ bị từ chối yêu cầu.

Chía khóa để làm việc này là mã thông báo CSRF cung cấp phải nằm trong một phần yêu cầu HTTP không được trình duyệt tự động đưa vào.

Ví dụ: request token CSRF cung cấp nằm trong request parameter hoặc header http sẽ được bảo vệ khỏi các cuộc tấn công CSRF.

Request token CSRF cung cấp nằm trong cookie không hoạt động vì cookie được trình duyệt tự động đưa vào request http.

Ta có thể cài đặt chỉ yêu cầu token CSRF cho các request cập nhật ứng dụng. Điều này cho phép liên kết đến trang web của mình từ các trang web bên ngoài.

Ngoài ra không nên đưa token vào Http Get vì điều này có thể khiến token bị rò rỉ.

Ví dụ: Cải tiến form chuyển tiền của ví dụ bên trên sử dụng CSRF

<form method=”post” action=”/transfer”>

<input type=”hidden” name=”\_csrf” value=”4bfd1575-3ad1-…”>

<input type=”text” name=”amount”>

<input type=”text” name=”routingNumber”>

<input type=”text” name=”account”>

<input type=”submit” name=”Transfer”>

</form>

Thuộc tính SameSite

Một cách mới để bảo vệ khỏi các cuộc tấn công CSRF là chỉ định thuộc tính SameSite trên cookie.

Server có thể chỉ định thuộc tính SameSite khi thiết lập cookie để quy định rằng cookie sẽ không được gửi khi request đến từ các trang web bên ngoài.

Spring Security không trực tiếp kiểm soát việc tạo cookie session, vì vậy nó không cung cấp hỗ trợ cho thuộc tính SameSite.

* **Spring Session** cung cấp hỗ trợ thuộc tính SameSite trong các ứng dụng trên Servlet.
* **CookieWebSessionIdResolver** cung cấp hỗ trợ thuộc tính SameSite cho các ứng dụng WebFlux.

Ví dụ: header phản hồi request http với thuộc tính SameSite có thể trông giống như sau:

Set-Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly; SameSite=Lax

Các giá trị hợp lệ cho thuộc tính SameSite là:

* Strict: Khi được chỉ định, bất kỳ request nào đến từ cùng một trang web sẽ bao gồm cookie. Nếu không, cookie sẽ không được đưa vào request http.
* Lax: Khi được chỉ định, các request nào đến từ cùng một trang web, hoặc request đến từ các điều hướng cấp cao nhất và phương thức Idempotent. Nếu không, cookie sẽ không được đưa vào request http.

Một số cân nhắc khi sử dụng:

Đặt thuộc tính SameSite ở chế độ strict cung cấp phòng thủ chiều sau, chống lại các cuộc tán công của CSRF, nhưng có khả năng gây nhầm lẫn cho người dùng trừ khi các nhà phát triển của trang web đảm bảo rằng hệ thống quản lý transaction hợp lý với cấp cao nhất điều hướng.

Xem xét trường hợp người dùng đọc email của họ tại MegaCorp.

Nhà cung cấp email trực tuyến của Inc ‘https://example.com/’. Họ có thể mong đợi, nhấn vào một liên kết được gửi qua email tới ‘https://projects.com/serect’. Nhưng nếu “projects.com” đánh dấu cookie session của họ là SameSite 🡪 không thể điều hướng đến trang web này được nữa.

Các nhà phát triển có thể tránh nhầm lẫn này bằng cách áp dụng quản lý cookie session dựa vào 2 cookie:

* 1 để đọc dữ liệu 🡪 không khai báo thuộc tính SameSite
* 1 để ghi dữ liệu 🡪 khai báo thuộc tính SameSite

Còn 1 cách đó là: trước đây có thể chọn chế độ “Lax” và điều hướng cấp cao nhất để cho phép người dùng điều hướng đến trang web của mình

**Lưu ý:** hầu hết các trình duyệt hiện tại đều hỗ trợ thuộc tính SameSite.

Khi nào nên sử dụng bảo vệ CSRF

**Bảo vệ CSRF với Json**

Một câu hỏi phổ biến là “tôi có cần bảo vệ các yêu cầu Json do javascript thực hiện không” 🡪 câu trả lời: là còn phụ thuộc vào nhiều yếu tố.

Tuy nhiên, ta cần phải cẩn thận vì có những cách khai thác lỗ hổng bảo mật CSRF thông qua các request chứa Json.

Ví dụ: hacker có thể tạo CSRF với Json bằng cách sử dụng biểu mẫu sau;

<form action=”<https://bank.example.com/transfer>” method=”post” enctype=”text/plain”>

<input name=’{“amount”:100, “routingNumber”:”evilsR”, “account”:”evils”, “**ignore\_me**”: “’ value=’test”}’ type=’hidden’>

<input type=”submit” value=”Win money!”>

</form>

Điều này sẽ tạo ra cấu trúc json sau:

{

“amount”: 100,

“routingNumber”: “evilsR”,

“account”: “evils”,

“ignore\_me”: “=test”

}

Nếu một ứng dụng không chỉ rõ loại Content-Type 🡪 lỗ hổng này sẽ bị khai thác.

Tùy thuộc vào thiết lập, ứng dụng Spring MVC chỉ rõ loại Content-Type vẫn có thể bị khai thác lỗ hổng bằng cách kết thúc URL bằng **.json** như ví dụ sau:

<form action=”[https://bank.example.com/**transfer.json**](https://bank.example.com/transfer.json)” method=”post” enctype=”text/plain”>

Tương tự như trên.

</form>

Một số lưu ý đặc biệt cần xem xét khi triền khai bảo vệ chống lại CSRF

Phần này đọc sau.

**Đăng nhập –** không hiểu, xem xét sau.

**Đăng xuất –** không hiểu, xem xét sau.

**CSRF và thời gian chờ session –** xem xét sau.

**Multipart (tải tệp lên) –** xem xét sau.

## Header HTTP

Có nhiều header response HTTP có thể được sử dụng để tăng tính bảo mật của các ứng dụng web. Phần này dành riêng cho các header response HTTP khác nhau mà spring security cung cấp hỗ trợ rõ ràng. Nếu cần, Spring Security cũng có thể được định cấu hình để cung cấp các header tùy chỉnh.

### Tiêu đề bảo mật mặc định

Spring Security cung cấp một bộ header response HTTP mặc định liên quan đến bảo mật để cung cấp các giá trị mặc định an toàn.

Header Spring Security mặc định như sau:

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

X-Content-Type-Options: nosniff

Strict-Transport-Security: max-age: 31536000; includeSubDomains

X-Frame-Options: DENY

X-XSS-Protection: 1; mode=block

Strict-Transport-Security chỉ được thêm vào các request https.

**Cache-Control**

Mặc định Spring Security là tắt bộ nhớ đệm để bảo vệ nội dung của người dùng.

Nếu người dùng xác thực để xem thông tin nhạy cảm và sau đó đăng xuất, ta sẽ không muốn “kẻ trộm” **bấm quay lại** dể xem thông tin nhạy cảm. Các header kiểm soát bộ nhớ cache mặc định là:

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

Để được bảo mật theo mặc định, Spring Security thêm các header này theo mặc định. Tuy nhiên, nếu ứng dụng của ta cung cấp các header kiểm soát bộ nhớ cache của riêng nó thì **Spring Security sẽ không hoạt động**. Điều này cho phép các ứng dụng đảm bảo rằng các tài nguyên tĩnh như CSS và Javascript có thể được lưu vào bộ nhớ cache.

**Content-Type Options** – Tùy chọn nội dung

Các trình duyệt trước đây, bao gồm cả Internet Explorer, sẽ cố gắng đoán lại **Content-Type** bằng cách sử dụng tính năng **dò tìm** Content

* Điều này cho phép trình duyệt cải thiện trải nghiệm của người dùng bằng cách đoán loại Content trên tài nguyên chưa phân loại Content.
* Ví dụ: Nếu một trình duyệt gặp 1 **tệp** Javascript không có Content-Type được chỉ định, nó sẽ có thể đoán loại Content và sau đó chạy nó.

Có nhiều thức ta nên làm để phân loại nội dung như:

* Chỉ hiển thị tài nguyên trong một domain riêng biệt
* Đảm bảo Content-Type được đặt
* Santilize tài liệu

Tuy nhiên các biện pháp này nằm ngoài phạm vi của những gì Spring Security cung cấp.

Spring Security tắt tính năng dò tìm nội dung theo mặc định bằng cách thêm header sau vào response http

X-Content-Type-Options: nosniff

**Lưu ý:** việc cài đặt spring security cho phép đoán Content-Type, có thể tạo lỗ hổng cho các cuộc tấn công **XSS**.

### HTTP Strict Transport Security (HSTS)

Khi ta truy cập trang web của ngân hàng, ta nhập:

* **http**:mybank.example.com
* Hay **https**:mybank.example.com

Nếu ta không sử dụng giao thức https, ta có thể dễ dàng bị tấn công bởi Man In the Middle.

Ngay cả khi trang web thực hiện chuyển hướng đến https:mybank.example.com, người dùng độc hại có thể chặn yêu cầu HTTP ban đầu và thao túng response (ví dụ: chuyển hướng đến https:mibank.example.com và đánh cắp thông tin đagnư nhập của họ)

Nhiều người dùng bỏ qua giao thức https 🡪 đây chính là lý do HSTS được tạo ra.

Sau khi https:mybank.example.com được chỉ định làm máy chủ lưu trữ HSTS, trình duyệt có thể biết trước rằng bất kỳ yêu cầu nào đến http:mybank.example.com sẽ được hiểu là https:mybank.example.com

* Điều này làm giảm đáng kể khả năng xảy ra tấn công Man in the middle

Theo RFC6797, header HSTS chỉ được được vào các phản hổi https. Để trình duyệt xác nhận header, trước tiên trình duyệt phải tin tường CA đã ký chứng chỉ SSL được sử dụng để tạo kết nối.

Một cách để tang web được đánh dấu là máy chủ HSTS là tải máy chủ đó vào trình duyệt.

Cách khác là thêm Strict-Transport-Security vào header response.

Ví dụ: hành vi mặc định của Spring Security là thêm header sau để hướng dẫn trình duyệt coi miền như một máy chủ HSTS trong 1 năm (3153600):

Strict-Transport-Security: max-age=3153600; includeSubDomains; preload

* Lệnh tùy chọn includeSubDomans hướng dẫn trình duyệt rằng miền phụ (ví dụ: <http://secure.mybank.example.com>) cũng phải được coi là miền HSTS
* Lệnh tùy chọn preload hướng dẫn trình duyệt rằng miền phải được tải trước trong trình duyệt dưới dạng miền HSTS.

### Http public Key pining (HPKP)

Để duy trì trạng thái thụ động, Spring Security vẫn cung cấp hỗ trợ cho HPKP trong mỗi trường servlet, nhưng vì những lý do được liệt kê ở phía dưới, HPKP không còn được nhóm bảo mật khuyến nghị nữa.

HPKP chỉ định cho client public key nào sẽ được sử dụng với 1 số server nhất định để ngăn các cuộc tấn công Man in the Middle với các chứng chỉ giả mạo. Khi được sử dụng đúng cách, HPKP có thể thêm vào các lớp bảo vệ bổ sung chống lại chứng chỉ bị xâm phạm. Tuy nhiên do sự phức tạp của HPKP, nhiều chuyên gia không còn khuyến khích sử dụng nó nữa, Chrome thậm chí đã loại bỏ hỗ trợ cho nó.

### X-Frame-Options

Cho phép trang web của bạn được thêm vào khung (frame) có thể là một vấn đề bảo mật.

Ví dụ: Sử dụng kiểu CSS thông minh, người dùng có thể bị lừa nhấp vào một thứ gì đó mà họ không có ý định nhấn. Như một người dùng đã đăng nhập vào ngân hàng của họ, có thể nhấp vào nút cấp quyền truy cập cho những người dùng khác. Kiểu tấn công này được gọi là Clickjacking.

Một cách tiếp cận hiện đại khác để đối phó với Clickjacking là sử dụng Chính sách bảo mật nội dung (CSP)

Có một số cách để giảm thiểu các cuộc tấn công Clickjacking. Ví dụ: Để bảo vệ các trình duyệt cũ khỏi các cuộc tấn công clickjacking, ta có thể sử dụng **mã phá khung** (frame breaking node). Mặc dù không hoàn hảo, nhưng mã phá khung là cách tốt nhất ta có thể làm cho các trình duyệt cũ.

Một cách tiếp cận hiện đại hơn để giải quyết Clickjacking là sử dụng header X-Frame-Options. Theo mặc dịnh, Spring Security **vô hiệu hóa**  các trang hiển thị trong iframe, bằng header sau:

X-Frame-Options: DENY

### Bảo vệ X-XSS

Một số trình duyệt đã tích hợp hỗ trợ để lọc ra các cuộc tấn công XSS được phản ánh. Điều này không có nghĩa là dễ hiểu ???, nhưng hỗ trợ trong việc bảo vệ khỏi XSS.

Tính năng lọc thường được bật theo mặc định, vì vậy việc thêm header vào request chỉ đảm bảo nó được bật và hướng dẫn trình duyệt phải làm gì khi phát hiện thấy một cuộc tấn công XSS.

Ví dụ: bộ lọc có thể cố gắng thay đổi nội dung theo cách ít xâm lấn nhất để vẫn hiển thị đầy đủ thông tin.

Đôi khi, kiểu thay thế này có thể trở thành một lỗ hổng XSS. Thay vào đó, tốt nhất là **chặn nội dung**, hơn là cố gắng **thay đổi** nó.

Theo mặc định, Spring Security chặn nội dung bằng cách sử dụng header sau:

X-XSS-Protection: 1; mode=block

### Chính sách bảo mật nội dung (CSP)

Chính sách bảo mật nội dung CSP là một cơ chế mà các ứng dụng Web có thể tận dụng để giảm thiểu các lỗ hổng chèn nội dung (**injection** content), chẳng hạn như cross-site scripting (XSS). CSP là một chính sách khai bào cung cấp cơ sở cho các nhà phát triển ứng dụng web khai bào và cuối cùng **thông báo** cho người dùng về **các nguồn** mà **web định tải tài nguyên**.

**Lưu ý**: Chính sách bảo mật nội dung không nhằm giải quyết tất cả các lỗ hổng chèn nội dung. Thay vào đó, CSP có thể được tận dụng để giúp giảm tác hại do các cuộc tấn công chèn nội dung. Là tuyến phòng thủ đầu tiên, các nhà phát triển nên **xác nhận đầu vào** của họ và **mã hóa** **đầu ra**.

Ứng dụng Web có thể sử dụng CSP bằng cách bao gồm 1 trong các header sau trong phản hồi:

* Content-Security-Policy
* Content-Security-Policy-Report-Only

Một header này được sử dụng như 1 cơ chế để cung cấp CSP cho client. Chính sách bảo mật chứa một tập hợp các chỉ thị, mỗi **chỉ thị** chịu trách nhiệm khai báo các hạn chế cho một biểu diễn tài nguyên cụ thể.

Ví dụ: một ứng dụng web có thể khai báo rằng nó mong đợi tải các tập lệnh từ nguồn cụ thể, đáng tin cậy, bằng cách bao gồm header sau trong phản hồi:

Content-Sercurity-Policy: script-src <https://trustedscripts.example.com>

Nếu trang web cố gắng tải một tập lệnh từ một nguồn khác với những gì được khai báo trong **chỉ thị** script-src sẽ bị chặn bởi client. Ngoài ra, nếu chị thị report-uri được khai báo trong chính sách bảo mật, thì hành vi vi phạm sẽ được tác nhân người dùng báo cáo tới Url đã khai báo.

Ví dụ: Nếu một ứng web vi phạm chính sách bảo mật đã khai báo, header sau sẽ hướng dẫn client gửi báo cáo vi phạm đến Url được chỉ định trong report-uri

Content-Security-Policy: script-src <https://trusted.example.com>; report-uri /csp-report-end-point/

Báo cáo vi phạm là dữ liệu Json, có thể được thu thập bởi api riêng của ứng dụng web hoặc bởi dịch vụ váo cáo vi phạm CSP được lưu trữ công khai, chẳng hạn như report-uri.com

Header Content-Security-Policy-Report-Only cung cấp khả năng cho các admin của ứng dụng web giám sát các chính sách bảo mật, thay vì thực thi chúng. Header này thường được sử dụng khi thử nghiệm hoặc phát triển các chính sách bảo mật cho một trang web. Thay vào đó, khi 1 chính sách được coi là có hiệu lực, nó có thể được thực thi bằng cách sử dụng header Content-Security-Policy.

Ví dụ: với header response sau, chính sách tuyên bố rằng cách tập lệnh có thể được tải từ một trong 2 nguồn:

Content-Security-Policy-Report-Only: script-src ‘self’ <https://trusted.example.com>; report-uri /csp-report-endpoint/

Nếu trang web vi phạm chính sách này, bằng cách cố gắng tải tài nguyên từ trang web khác (chẳng hạn: evil.com), client sẽ gửi báo cáo vi phạm đến Url đã được chỉ định bới report-uri, nhưng vẫn cho phép tải tài nguyên vi phạm.

### Chính sách giới thiệu – Referrer Policy

Chính sách liên kết giơi thiệu là một cơ chế mà cac ứng dụng Web có thể tận dụng để quản lý trường liên kết giới thiệu, trường này chứa trang cuối cùng mà người dùng đã truy cập

Spring Security là sử dụng header để khai báo:

Referrer-Policy: same-origin

Heade response Referrer-Policy hướng dẫn trình duyệt cho đích biết nguồn gửi thông tin đến là gì.

### Chính sách tính năng - Feature Policy

Feature Policy là một cơ chế cho phép các nhà phát triển web bật, tắt và sửa đổi 1 cách có chọn lọc hoạt động của các api và tính năng web nhất định trong trình duyệt.

Ví dụ:

Feature-Policy: geolocation ‘self’

Với chính sách tính năng, nhà phát triển có thể chọn tham gia một tập hợp các “chính sách” để trình duyệt thực thi trên các tính năng cụ thể được sử dụng trên toàn bộ trang web của ta. Các chính sách này hạn chế những api nào mà trang web có thể truy cập hoặc sửa đổi hành bị mặc định của trình duyệt đối với 1 số api nhất định.

### Chính sách quyền – Permission-Policy

Chính sách quyền là một cơ chế cho phép nhà phát triển bật, tắt và sửa đổi một cách có chọn lọc hoạt động của các API và tính năng web nhất định trong trình duyệt.

Ví dụ:

Permission-Policy: geolocation=(self)

### Xóa dữ liệu trang web – Clear site data

Xóa dữ liệu trang web là một cơ chế mà theo đó bất kỳ dữ liệu phía trình duyệt nào –cookie, cache và những thứ tương tự - đều bị xóa khi response có chứa header này:

Clear-Site-Data: “cache”, “cookies”, “storage”, “executionContexts”

Đây là một hành động dọn dẹp tốt để thể thực hiện khi đăng xuất.

### Chính sách đa nguồn gốc – Cross-Origin Policies

Spring Security cung cấp hỗ trợ cho một số header quan trọng của Cross-Origin Policies. Những header đó là:

* Cross-Origin-Opener-Policy
* Cross-Origin-Embedder-Policy
* Cross-Origin-Resource-Policy

Cross-Origin-Opener-Policy (COOP): ngăn chặn bất kỳ truy cập DOM trực tiếp nào giữa các nguồn khác nhau

Cross-Origin-Embedder-Policy (COEP): ngăn chặn các nguồn không rõ nguồn gốc được tải tài nguyên.

Cross-Origin-Resource-Policy (CORP): cho phép ta kiểm soát tập hợp các nguồn được trao quyền, được phép tải tài nguyên. Nó là một biện pháp phòng thủ chắc chắn chống lại các cuộc tấn công như Spectre, vì nó cho phép các trình duyệt chặn một phản hồi nhất định trước khi bị tấn công.

## Request HTTP

Tất cả giao tiếp dựa trên HTTP, bao gồm tài nguyên tĩnh, phải được bảo vệ bằng TLS.

Là một khuôn khổ, Spring Security không xử lý các kết nối HTTP, do đó không cung cấp hỗ trợ trực tiếp cho HTTPS. Tuy nhiên nó cung cấp 1 số tính năng giúp sử dụng HTTPS.

### Chuyển hướng sang HTTPS

Khi client sử dụng HTTP, Spring Security có thể được cấu hình để chuyển hướng đến HTTPS trong cả môi trường Servlet và WebFlux.

### Strict Transport Security

Spring Security cung cấp tính năng này mặc định.

### Proxy Server Configuration

Khi sử dụng server proxy, điều quan trọng là đảm bảo rằng ta đã cấu hình ứng dụng của mình đúng cách.

Ví dụ: Nhiều ứng dụng sẽ có load balance phản hồi yêu cầu là <https://example.com> bằng cách chuyển tiếp yêu cầu đến máy chủ tại: <https://192.168.1:8080>. Nếu không có cấu hình thích hợp, server sẽ không biết rằng bộ cân bằng tải tồn tại và xử lý yêu cầu như thể client yêu cầu <https://192.168.1:8080>.

Để khắc phục điều này, ta có thể sử dụng RFC-7239 để chỉ định rằng bộ cân bằng tải đang được sử dụng. Để làm cho ứng dụng nhận biết được điều này, ta cần phải định cấu hình server, nhận biết được header **X-Forwarded**.

Ví dụ: Tomcat sử dụng RemotelpValve và Jetty sử dụng ForwardedRequestCustomizer. Ngoài ra, người dùng Spring có thể tận dụng ForwardedHeaderFilter

Người dùng Spring Boot có thể sử dụng thuộc tính **server.use-forward-headers** để cấu hình ứng dụng.

## Tích hợp

Spring Security cung cấp tích hợp với nhiều khung (frame) và Api. Trong phần này ta sẽ tìm hiểu về tích hợp chung không dành tiền cho môi trường Servlet hoặc Reactive

## Spring Security Crypto Module

### Giới thiệu

Module Spring Sercurity Crypto cung cấp hỗ trợ mã hóa đối xứng, tạo khóa và mã hóa mật khẩu. Mã được phân phối như một phần của module lỗi nhưng không phụ thuộc và bất kỳ mã Spring Security (hoặc Spring) nào khác.

### Bộ mã hóa – Encryptors

Lớp Encryptors cung cấp các phương thức gốc để xây dựng các bộ mã hóa đối xứng. Sử dụng lớp này, ta có thể tạo:

* ByteEncryptors để mã hóa dữ liệu ở dạng byte[].
* TextEncryptors để mã hóa chuỗi văn bản.

Bộ mã hóa đồng bộ theo luồng (thread).

**BytesEncryptor**

Sử dụng phương thức Encryptors.stronger để tạo BytesEncryptor:

Ví dụ:

Encryptors.stronger(“<password>”, “<salt>”)

Phương pháp mã hóa “mạnh hơn” tạo ra một bộ mã hóa sử dụng mã hóa AES 256 bit với chế độ bộ đếm Galois (GCM). Nó lấy ra secret key bằng cách sử dụng PBKDF2 của PKCS #5 (Chức năng lấy khóa dựa trên “<password>” #2). Phương pháp này yêu cầu Java 6.

* “<password>” được sử dụng để tạo secret key phải được giữ ở nơi an toàn và không được chia sẻ.
* “<salt>” được sử dụng để ngăn chặn các cuộc tấn công từ điển đối với khóa, trong trường hợp dữ liệu mã hóa của ta bị đánh cắp. <salt> được cung cấp phải ở dạng chuỗi mã hex, là ngẫu nhiên và có độ dài ít nhất 8 bytes. Mỗi <salt> như vậy có thể được tạo ra bằng cách sử dụng KeyGenerator:

String salt = KeyGenerators.string().generateKey

* Một vector khởi tạo ngẫu nhiên 16 byte cũng được áp dụng để mỗi thông điệp được mã hóa là duy nhất

Người dùng cũng có thể sử dụng phương pháp mã hóa standard, là AES 256-bit trong chế độ chuỗi khối mã hóa (CBC). Chế độ này không được xác thực và không cung cấp bất kỳ đảm bảo nào về tính xác thực của dữ liệu.

**TextEncryptor**

Sử dụng phương thức Encryptors.text để tạo một TextEncryptor:

Ví dụ:

Encryptors.text(“<password>”, “<salt>”)

TextEncryptor sử dụng BytesEncryptor tiêu chuẩn để mã hóa dữ liệu văn bản. Kết quả được mã hóa được trả về dưới dạng mã hex, để dễ dàng lưu trữ trên hệ thống tệp hoặc trong CSDL.

Sử dụng phương thức gốc Encryptors.queryableText để tạo TextEncryptor “có thể truy vấn”:

Ví dụ:

Encryptors.queryableText(“<password>”, “<salt>”)

Sự khác biệt giữa TextEncryptor có thể truy vấn và TextEncryptor tiêu chuẩn liên quan đến việc xử lý vector khởi tạo. vector khởi tạo được sử dụng trong hoạt động mã hóa TextEncryptor có thể truy vấn, được chia sẻ hoặc không đổi và không được tạo ngẫu nhiên. Điều này có nghĩa là một văn bản được mã hóa nhiều lần sẽ luôn tạo ra cùng một kết quả mã hóa. Điều này kém an toàn hơn, nhưng cần thiết truy vấn nhiều lần dữ liệu được mã hóa. Một ví dự về văn bản được mã hóa có thể ttruy vẫn sẽ là apiKey Oauth.

### Key Generators

Lớp KeyGenerators cung cấp một số phương thức để tạo khóa chính khác nhau. Sử dụng lớp này ta có thể tạo:

* BytesKeyGenerator để tạo các khóa byte[]
* StringKeyGenerator để tạo các khóa chuỗi

KeyGenerators là một thread synchronized

**BytesKeyGenerator**

Mặc định – độ dài khóa là 8 byte:

BytesKeyGenerator generator = KeyGenerators.secureRandom();

byte[] key = generator.generateKey();

Tùy chọn độ dài khóa:

KeyGenerators.secureRandom(16)

Trả về cùng 1 khóa cho mọi lần gọi + tùy chọn độ dài:

KeyGenerators.shared(16)

**StringKeyGenerator**

Khởi tạo 🡪 để tạo ra khóa chính gồm 8 byte, mã hóa hex

KeyGenerators.string()

### Password Encoding

Gói mật khẩu của module spring-security-crypto cung cấp hỗ trợ cho việc mã hóa mật khẩu. PasswordEncoder là interface root và có dạng như sau:

public interface PasswordEncoder :

String encode (CharSequence rawPassword);

boolean matches (CharSequence rawPassword, String encodedPassword);

default boolean upgradeEncoding(String encodedPassword): return false;

Phương thức matchers trả về true nếu rawPassword, sau khi được mã hóa = encodedPassword

## Spring Data Integration

Spring Security cung cấp tích hợp Spring Data cho phép đề cập đến người dùng hiện tại trong các truy vấn của ta.

### Spring Data và Spring Security Configuration

Bổ sung thêm dependency:

org.springframework.security:spring-security-data

sẽ cung cấp thêm loại bean SecurityEvaluationContextExtension để ta **khai báo**.

@Bean

public SecurityEvaluationContextExtension securityEvaluationContextExtension() {

return new SecurityEvaluationContextExtension();

}

### Biểu thức bảo mật trong @Query

Bây giờ Spring Security có thể được sử dụng trong các truy vấn.

Ví dụ:

@Repository

public interface MessageRepository extends PagingAndSortingRepository<Message,Long> {

@Query("select m from Message m where m.to.id = ?#{ principal?.id }")

Page<Message> findInbox(Pageable pageable);

### Concurrency Support

Trong hầu hết các môi trường, Security được lưu trữ trên Thread chính. Điều đó có nghĩa là khi công việc được thực hiện trên một Thread mới, Thread này sẽ không có SecurityContext. Do vây:

* Spring Security cung cấp một cơ sở hạ tầng để giúp ngừi dùng thực hiện điều này dễ dàng hơn nhiều.
* Spring Security cung cấp các bản tóm tắt mức thấp để làm việc với Spring Security trong môi trường đa luồng.
* Trên thực tế, đây là những gì Spring Security xây dựng để tích hợp với AsyncContext.start(Runnable) và Spring MVC Async Integration

**DelegatingSecurityContextRunnable**

Một trong những nền tảng cơ bản nhất trong concurrency support của Spring Security là DelegateSecurityContextRunnable. Nó bao bọc một Runnable để khởi tạo SecurityContextHolder với 1 SecurityContext. Sau đó, nó gọi Runnable ủy quyền để đảm bảo xóa SecurityContextHolder sau đó.

Trong DelegatingSecurityContextRunnable giống như sau:

public void run() {

try {

SecurityContextHolder.setContext(securityContext);

delegate.run();

} finally {

SecurityContextHolder.clearContext();

}

}

Mặc dù rất đơn giản nhưng nó giúp chuyển SecurityContext từ Thread này sang Thread khác. Điều này rất quan trọng vì rong hầu hết các trường hợp, SecurityContextHolder hoạt riêng lẻ trên từng Thread.

Ví dụ: ta có thể sử dụng hỗ trợ <global-method-security> của Spring Security để bảo mật một trong các dịch vụ của mình. Bây giờ ta có thể dễ dàng chuyển SecurityContext từ Thread này sang Thread khác.

Dưới đây là một ví dụ về cách ta có thể làm điều này:

Runnable originalRunnable = new Runnable() {

public void run() {

// invoke secured service

}

};

SecurityContext context = SecurityContextHolder.getContext();

DelegatingSecurityContextRunnable wrappedRunnable =

new DelegatingSecurityContextRunnable(originalRunnable, context);

new Thread(wrappedRunnable).start();

Đoạn code trên có thể viết tắt như sau:

Runnable originalRunnable = new Runnable() {

public void run() {

// invoke secured service

}

};

DelegatingSecurityContextRunnable wrappedRunnable =

new DelegatingSecurityContextRunnable(originalRunnable);

new Thread(wrappedRunnable).start();

**DelegatingSecurityContextExecutor**

Trong phần trước, ta đã thấy rằng rất dễ để sử dụng DelegatingSecurityContextRunnable, nhưng nó không phải lý tưởng vì ta phải biết Spring Security để sử dụng nó.

Ta hãy xem cách DelegatingSecurityContextExecutor có thể bảo vệ mã của ta khỏi bất kỳ sự hiểu biết nào về việc ta đang sử dụng Spring Security.

Thiết kế của DelegatingSecurityContextExecutor rất giống với thiết kế của DelegatingSecurityContextRunnable ngoại trừ nó chấp nhận Executor thay vì Runnable.

Ví dụ:

SecurityContext context = SecurityContextHolder.createEmptyContext();

Authentication authentication = UsernamePasswordAuthenticationToken.authenticated(“user”, “doesnotmatter”, AuthorityUtils.createAuthorityList(“ROLE\_USER”));

context.setAuthentication(authentication);

SimpleAsyncTaskExecutor delegateExecutor = new SimpleAsyncTaskExecutor();

DelegatingSecurityContextExecutor executor = new DelegatingSecurityContextExecutor(delegateExecutor, context);

Runnabale originalRunnable = new Runnable () {

public void run () {  
 // invoke secured service

}

}

executor.execute(originalRunnable)

🡺 Tại thời điểm này, ta có thể tự hỏi:” Làm thế nào điều này bảo vệ mã của ta về bất kỳ kiến thức nào về Spring Security?”

TL: Bởi vì thay vì tạo SecurityContext và DelegatingSecurityContextExecutor trong mã riêng của mình, ta có thể đưa vào một phiên bản đã được khởi tạo của DelegatingSecurityContextExecutor.

@Autowired

private Executor executor; // becomes an instance of our DelegatingSecurityContextExecutor

public void submitRunnable() {

Runnable originalRunnable = new Runnable() {

public void run() {

// invoke secured service

}

};

executor.execute(originalRunnable);

}

Bây giờ mã của ta không biết rằng cái SecurityContext đang được truyền tới Thread, sau đó cái originalRunnable được chạy, và sau đó SecurityContextHolder bị xóa.

Trong ví dụ này, tất cả các luồng đều sử dụng chung một người dùng hay SecurityContext.

Điều gì sẽ xảy ra nếu ta muốn sử dụng người dùng SecurityContextHolder tại thời điểm ta gọi executor.execute(Runnable) (tức là người dùng hiện đang đăng nhập) để xử lý originalRunnable? Điều này có thể thực hiện bằng cách xóa đối số SecurityContext khỏi hàm DelegatingSecurityContextExecutor của ta.

Ví dụ:

SimpleAsyncTaskExecutor delegateExecutor = new SimpleAsyncTaskExecutor();

DelegatingSecurityContextExecutor executor =

new DelegatingSecurityContextExecutor(delegateExecutor);

Bây giờ bất cứ lúc nào executor.execute(Runnable) được thực hiện, trước tiên SecuriryContext bị lấy bởi SecuriryContextHolder và sau đó SecurityContext được sử dụng để tạo DelegatingSecurityContextRunnable. Điều này nghĩa là ta đang chạy Runnable với cùng một người dùng đã được sử dụng để gọi mã executor.execute(Runnable).

**Tất cả các class của Spring Security Concurrency**

* DelegatingSecurityContextCallable
* DelegatingSecurityContextExecutor
* DelegatingSecutiryContextExecutorService
* DelegatingSecurityContextRunnable
* DelegatingSecurityContextScheduledExecutorService
* DelegatingSecurityContextSchedulinhTaskExecutor
* DelegatingSecurityContextAsyncTaskExecutor
* DelegatingSecurityContextTaskExecutor
* DelegatingSecurityContextTaskScheduler.

### Hỗ trợ Jackson

Spring Security cung cấp hỗ trợ Jackson cho các lớp liên quan đến Spring Security. Điều này giúp cải thiện hiệu suất tuần tự hóa (serializing) các lớp liên quan đến Spring Security khi làm việc với các phiên phân tán (tức là sao chép session, Spring session, …)

Đề sử dụng nó, hãy đăng ký SecurityJackson2Modules.getModules(ClassLoader) với ObjectMapper:

ObjectMapper mapper = new ObjectMapper()

ClassLoader loader = this.getClass().getClassLoader()

List<Module> modules = SecurityJackson2Modules.getModules(loader)

mapper.registerModules(modules)

SecurityContext context = new SecurityContextImpl()

String json = mapper.writeValueAsString(context)

**Ghi chú:** Các module Spring Security sau cung cấp hỗ trợ Jackson:

* spring-security-core (CoreJackson2Module)
* spring-security-web (WebJackson2Module, WebServletJackson2Module, WebServerJackson2Module)
* spring-security-oauth2-client (OAuth2ClientJackson2Module)
* spring-security-cas (CasJackson2Module)

### Localization – Bản địa hóa.

Nếu ta cần hỗ trợ các ngôn ngữ khác, mọi thứ ta cần biết đều có trong phần này.

Tất cả các thông báo ngoại lệ có thể được bản địa hóa, bao gồm các thông báo liên quan đến lỗi xác thực và quyền truy cập bị từ chối (lỗi ủy quyền).

* Các trường hợp exception và thông báo ghi nhật ký tập trung vào các nhà phát triển hoặc nhà khai thác hệ thống (bao gồm: cung cấp thuộc tính không chính xác, vi phạm quy tắc interface, sử dụng trình tạo không chính xác, xác thực thời gian khởi động, ghi nhật ký cấp debug) **không được bản địa hóa** và thay đó là được mã hóa cứng bằng tiếng anh trong mã Spring Security.

Trong file spring-security-core-xx.jar, ta sẽ tìm thấy package org.springframework.security có chứa danh sách các tệp message.properties.

Điều này nên được tham chiếu bởi ApplicationContext, vì các lớp Spring Security triển khai interface MessageSourceAware của Spring và mong đợi trình phân giải thông báo được đưa vào phụ thuộc vào thời gian khởi động ngữ cảnh. Thông thường tất cả những gì ta cần làm là đang ký một bean bên trong ngữ cảnh ứng dụng của bạn để tham chiếu đến thông báo.

Ví dụ:

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<property name="basename" value="classpath:org/springframework/security/messages"/>

</bean>

Tệp messages.properties mặc định là tiếng anh.

Nếu ta muốn tùy chình messages.properties hoặc hỗ trợ các ngôn ngữ khác, ta nên sao chép tệp, đổi tên tệp cho phù hợp và đăng ký tệp bên trong định nghĩa bean ở trên.

Để điều này hoạt động, ta cần phải đảm bảo rằng ngôn ngữ từ yêu cầu đến được lưu trữ trong org.springframework.context.i18n.LocaleContextHolder. Spring MVC DispatcherServlet sẽ tự động thực hiện điều này, **nhưng** vì các bộ lọc của Spring Security được thực hiện trước điều này, nên LocaleContextHolder cần phải thiết lập để chứa đúng Locale trước khi các bộ lọc được gọi. Ta có thể thực hiện điều này trong một bộ lọc (phải có trước bộ lọc Spring Security) hoặc ta có thể sử dụng Spring RequestContextFilter.

# Các module của dự án

Phần này lên trực tiếp tra cứu tại: [Project Modules and Dependencies :: Spring Security](https://docs.spring.io/spring-security/reference/modules.html)

# Ứng dụng Servlet

Spring Security tích hợp với Servlet Container bằng cách sử dụng bộ Filter của Servlet. Điều này có nghĩa, ta không cần phải sử dụng Spring, trong ứng dụng Servlet để sử dụng Spring Security.

## Định danh

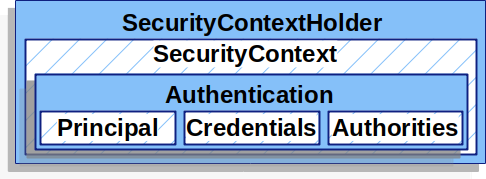
### Kiến trúc xác thực Servlet

Phần này, ta sẽ tìm hiểu về [Servlet Security: The Big Picture](https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-architecture), mô tả các thành phần kiến trúc chính của Spring Security được sử dụng trong xác thực Servlet. Chi tiết tại ‘[Cơ chế xác thực](https://docs.spring.io/spring-security/reference/servlet/authentication/index.html#servlet-authentication-mechanisms)’:

* `SecurityContextHolder`: Đây là nơi Spring Security lưu trữ thông tin về người dùng đã xác thực.
* `SecurityContext`: sinh ra từ `SecurityContextHolder`, chứa `Authentication` của người dùng đã xác thực.
* `Authentication`: cung cấp thông tin xác thực của người dùng, được sử dụng bởi `SecurityContext`.
* `GrantedAuthority`: interface thể hiện các quyền của người dùng.
* `AuthenticationManager`: interface chứa các phương thức quy định cách Spring Security thực hiện xác thực người dùng.
* Nó làm những việc như: cấu hình xác thực dựa trên nguồn nào (ví dụ: ldap, jdbc, …), sử dụng những thuộc tính nào để xác thực, …
* `ProviderManager`: các lớp implements `AuthenticationManager`. Được sử dụng phổ biến nhất.
* `AuthenticationEntryPoint`: được sử dụng để yêu cầu xác minh đăng nhập (nghĩa là khi người dùng truy cập vào website, nhưng chưa xác thực. Lớp này sẽ đóng vai trò chuyển hướng request đến trang login, …)
* `AbstractAuthenticationProcessingFilter`: đây là lớp cha của tất cả Filter, được sử dụng để chặn request của người dùng 🡪 tiến hành xác thực nhờ các lớp `ProviderManager`. (Hãy nghĩ nó giống Filter trong Servlet, mà nó là Filter như thế thật mà).

#### SecurityContextHolder

Trọng tâm của mô hình xác thực Spring Security là `SecurityContextHolder`.



Đây là `SecurityContextHolder` là nơi Spring Security lưu trữ thông tin chi tiết về người dùng đã xác thực. Spring Security không quan tâm nội dung của SercurityContextHolder là gì. Nếu SecurityContextHolder chứa giá trị, thì giá trị đó được sử dụng làm người dùng xác thực.

**Ví dụ**: Tạo người dùng xác thực trực tiếp

SecurityContext context = SecurityContextHolder.createEmptyContext();

Authentication authentication = new TestingAuthenticationToken(“username”, “password”, “ROLE\_USER”)

context.setAuthentication(authentication);

SercurityContextHolder.setContext(context);

**Ví dụ**: Truy cập người dùng được xác thực hiện tại

SercurityContext context = SercurityContextHolder.getContext();

Authentication authentication = context.getAuthentication();

String username = authentication.getName();

Object principal = authentication.getPrincipal();

Collection<? extends GrantAuthority> authorities = authentication.getAuthorities();

Theo mặc định, hàm `SecurityContextHolder` sử dụng một ThreadLocal để lưu trữ các thông tin này, có nghĩa là `SecurityContext` luôn có sẵn trong các phương thức chạy chung một luồng, ngay cả khi `SecurityContext` không được chuyển rõ ràng trong các đối số của phương thức. Sử dụng ThreadLocal theo cách này khá an toàn, nếu cẩn thận để xóa thread sau khi yêu cầu của principal (người dùng) hoàn thành.

Một số ứng dụng không hoàn toàn phù hợp để sử dụng ThreadLocal, vì cách chúng hoạt động cụ thể với các luồng.

Ví dụ: một ứng dụng khách Swing có thể muốn tất cả các luồng trong mày ảo Java sử dụng cùng một ngữ cảnh bảo mật. `SecurityContextHolder` có thể được định cấu hình khi khởi động, để chỉ cách ta muốn SecurityContext được lưu trữ.

* Đối với một ứng dụng độc lập, ta sẽ sử dụng SecurityContextHolder.MODE\_GLOBAL
* Các ứng dụng khác, có thể muốn có các luồng sinh ra bởi luồng bảo mật, cũng có các thông tin bảo mật tương tự. Điều này đạt được bằng cách sử dụng SecurityContextHolder.MODE\_INHERITABLETHREADLOCAL.
* Ta cũng có thể tùy chỉnh chế độ mặc định: SecurityContextHolder.MODE\_THREADLOCAL. Bằng cách gọi một số phương thức tĩnh trên `SecurityContextHolder`

#### SecurityContext

không có gì để tìm hiểu.

#### Authentication

`Authentication` được dùng với 2 mục đích sau:

1. Là đầu vào của `AuthenticationManager`, để cung cấp thông tin xác thực mà người dùng đã cung cấp nhằm mục đích xác thực. Khi được sử dụng trong trường hợp này `isAuthenticated()` trả về false.
2. Đầu ra của `SercurityContext`.

`Authentication` chứa:

1. `principal` - xác định người dùng. Chứa các thông tin như: tên người dùng, mật khẩu, … Là một đối tượng, ví dụ có thể sử dụng: `UserDetails`.
2. `credentials` - thường là mật khẩu. Trong nhiều trường hợp, nội dung của trường này sẽ bị xóa sau khi xác thực, để đảm bảo thông tin không bị rò rỉ.
3. `authorities` - Danh sách các `GrantedAuthority`, chứa các quyền của người dùng.

#### GrantedAuthority

Thông thường các đối tượng `GrantedAuthority` là các quyền trên **toàn ứng dụng**. Chúng không giành riêng cho một url nhất định nào. Spring Security có được thiết kế để xử lý yêu cầu phân quyền theo miền, nhưng ta không nên làm vậy, vì với dữ liệu người dùng lớn, bộ nhớ sẽ nhanh chóng hết (hoặc ít nhất khiến ta mất nhiều thời gian để xác thực người dùng). Đối với việc bảo mật theo miến, ta nên sự dụng các giải pháp khác.

#### AuthenticationManager

`AutheticationManager` chứa các api xác định cách bộ lọc của Spring Security thực hiện xác thực.

Giá trị `Authentication` được trả về sau đó, sẽ được đặt vào `SecurityContextHolder` bởi bộ điểu khiển (tức là các bộ lọc của Spring Security) đã gọi `AuthenticationManager`.

Ta có thể triển khai `AuthenticationManager` bằng bất cứ điều gì, nhưng cách triển khai phổ biến nhất là `ProviderManager`.

#### ProviderManager

`ProviderManager` là cách triển khai thường được sử dụng nhất của `AuthenticationManager`.

`ProviderManager` là một trong số các `AuthenticationProvider`.

Mỗi `AuthenticationProvider` sẽ có 3 trường hợp khi tiến hành xác thực:

1. Thành công
2. Thất bại
3. Không thể đưa ra quyết đinh 🡪 truyền xuống để `AuthenticationProvider` tiếp theo xử lý.

🡺 Nếu không có bất kỳ `AuthenticationProvider` nào xác thực thành công 🡪 trả ra ngoại lệ `ProviderNotFoundException`.

🡺 Mỗi `AuthenticationManager` hỗ trợ 1 loại `Authentication` (ví dụ LdapAuthentication, JdbcAuthentication, …). Nếu khác loại hỗ trợ 🡪 trả ra ngoại lệ `AuthenticationException`.

Trong thực tế, mỗi `AuthenticationProvider` thực hiện một loại xác thực cụ thể.

Ví dụ:

* 1 `AuthenticationProvider` thực hiện xác thực tên người dùng / mật khẩu.
* 1 `AuthenticationProvider` thực hiện xác thực SAML.

Theo mặc định, `ProviderManager`, sẽ cố gắng xóa mọi thông tin bảo mật nhạy cảm khỏi `Authentication` trả về khi yêu cầu xác thực thành công. Điều này ngăn thông tin như mật khẩu được lưu trữ lâu hơn mức cần thiết trong `HttpSession`.

Điều này có thể gây ra sự cố khi ta cố sử dụng bộ nhớ cache để lưu thông tin người dùng. Ví dụ: Nếu `Authentication` chứa một tham chiếu đến một đối tượng trong cache (chẳng hạn UserDetails) và đối tượng đã bị xóa mật khẩu khi `Authenticaion` trả về 🡪 không thể xác thực. Để khắc phục vấn để này, ta có 2 cách:

1. Lưu đối tượng tham chiếu (ví dụ UserDetails) vào cache trước khi `Authentication` trả về.
2. Tắt thuộc tính `eraseCredentialsAfterAuthenticaiton` tên `ProviderManager`.

#### Yêu cầu thông tin đăng nhập với AuthenticationEntryPoint

`AuthenticationEntryPoint` được sử dụng để gửi phản hồi yêu cầu HTTP, thông tin xác thực từ client.

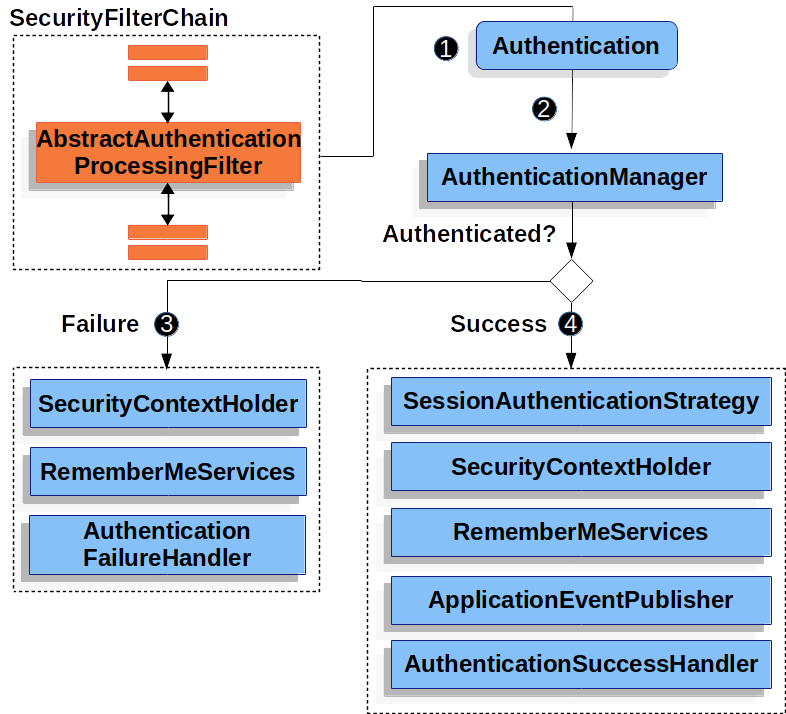
Đôi khi, người dùng sẽ chủ động bao gồm thông tin xác thực như tên người dùng / mật khẩu để yêu cầu tài nguyên. Trong trường hợp này Spring Security không cẩn cung cấp phản hồi HTTP yêu cầu thông tin xác thực từ khách hàng vì chúng đã được đưa vào.

Trong trường hợp khác, server sẽ đưa ra một yêu cầu, chứa thông tin những loại tài nguyên mà server muốn người dùng cấp quyền truy cập vào. Trong trường hợp này, tiển khai của `AuthenticationEntryPoint` được sử dụng để yêu cầu thông tin xác thực từ khách hàng. Việc triển khai `AuthenticationEntryPoint` có thể thực hiện chuyến hướng đến trang đăng nhập, phản hồi bằng tiêu đề WWW-Authentication, …

#### AbstractAuthenticationProcessingFilter

`AbstractAuthenticationProcessingFilter` là Filter cơ sở để xác thực thông tin đăng nhập của người dùng. Trước khi có thể xác thực thông tin đăng nhập, Spring Security thường yêu cầu thông tin đăng nhập bằng cách sử dụng AuthenticationEntryPoint.

Tiếp theo, `AbstractAuthenticationProcessingFilter` có thể xác thực bất kỳ yêu cầu xác thực nào gửi đến nó.



1. Khi người dùng gửi thông tin đăng nhập của họ, thì `AbstractAuthenticationProcessingFilter` sẽ tạo ra một `Authentication` từ HttpServeltRequest sẽ được xác thực. Loại `Authentication` được tạo phụ thuộc vào lớp còn của `AbstractAuthenticationProcessingFilter`. Ví dụ `UsernamePasswordAuthenticationFilter` sẽ tạo ra một `UsernamePasswordAuthenticationToken` tên người dùng và mật khẩu được gửi trong HttpServletRequest.
2. `Authentication` được chuyển vào `AuthenticationManager` để xác thực.
3. Nếu xác thực không thành công 🡪 request không thành công:

* `SercurityContextHolder` bị xóa.
* `RememberMeServices.loginFail` được gọi. Nếu remember me không được định cấu hình, thì đây là không cần thiết.
* `AuthenticationFailureHandler` được gọi.

1. Nếu xác thực thành công 🡪 request được đi tiếp.

* `SessionAuthenticationStrategy` được thông báo về một đăng nhập mới.
* `Authentication` sẽ được đặt vào `SecurityContextHolder`. Sau đó `SecurityContextPersistenceFilter` tiêm `SecurityContext` vào cho `HttpSession`.
* `RememberMeServices.loginSuccess` được gọi. Nếu remember me không được cấu hình, thì đây không cần thiết.
* `AuthenticationEventPublisher` xuất bản một `InteractiveAuthenticationSuccessEvent`
* `AuthenticationSuccessHandler` được gọi.

### Tên người dùng / mật khẩu.

#### Đọc tên người dùng và mật khẩu.

**Form login**

Spring Security cung cấp hỗ trợ cho tên người dùng và mật khẩu được cung cấp thông qua một biểu mẫu html.

Trang biểu mẫu login html: GET /login.

Trang đăng nhập của Spring Security được bật theo mặc định. Trang sẽ bị tắt nếu người dùng cung cấp một trang đăng nhập khác.

Ví dụ: Trang đăng nhập mặc định của Spring Security:  
public SecurityFilterChain filterChain (HttpSecurity http) {

http.formLogin (withDefaults());

}

Ví dụ: Trang đăng nhập tùy chỉnh:

public SecurityFilterChain filterChain(HttpSecurity http) {

http.formLogin (form -> form

.loginPage(“/login”)

.permitAll()

);

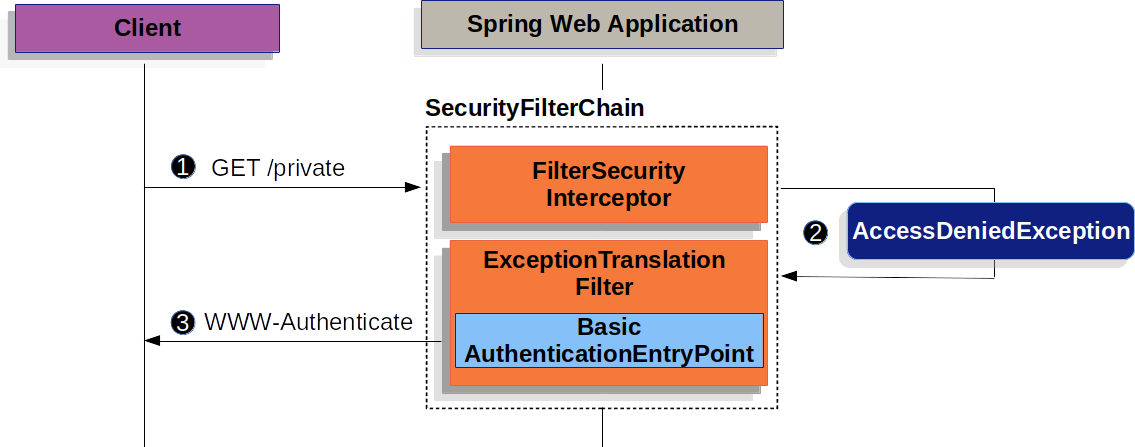
}

Nếu ta sử dụng Spring MVC, ta sẽ cần một @GetMapping(“/login”).

**Basic Authentication**

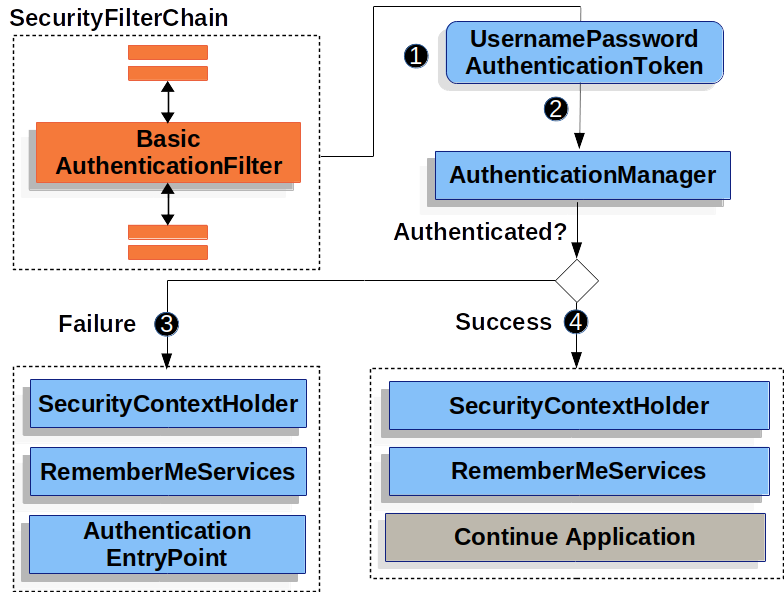
Phần này cung cấp chi tiết về cách Spring Security cung cấp hỗ trợ xác thực HTTP cơ bản cho các ứng dụng dựa trên servlet.

Hãy xem cách xác thực HTTP cơ bản hoạt động trong Spring Security. Đầu tiên, ta sẽ thấy header WWW-Authenticate được gửi trở lại một máy khách chưa được xác thực.



1. Đầu tiên, người dùng đưa ra một yêu cầu chưa được xác thực với tài nguyên /private mà nó không được ủy quyền.
2. `FilterSecurityInterceptor` chỉ ra rằng yêu cầu chưa được xác thực bị từ chối bằng cách nén ngoại lệ `AccessDeniedException`
3. Vì người dùng không được xác thực, sử dụng `ExceptionTranslationFilter` để bắt đầu xác thực. Triển khai của `AuthenticationEntryPoint` là `BasicAuthenticationEntryPoint` gửi header `WWW-Authentication`. `RequestCache` thường là một `NullRequestCache` không lưu yêu cầu vì client có khả năng phát lại các yêu cầu mà nó đã yêu cầu ban đầu.

Khi Client nhận được header `WWW-Authenticate`, nó biết rằng nên thử lại bằng tên người dùng và mật khẩu. Dưới đây là quy trình cho tên người dùng và mật khẩu đang được xử lý:



Ví dụ: Ta có thể cấu hình HTTP cơ bản như sau:

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) {

http.httpBasic (withDefaults());

return http.build();

}

#### Digest Authentication – Xác thực thông báo.

**Lưu ý**: Ta không nên sử dụng Digest Authentication trong các ứng dụng hiện tại vì nó không được coi là an toàn. Vấn đề rõ ràng nhất ở đây là ta phải lưu trữ mật khẩu của mình ở định dạng văn bản rõ ràng hoặc được mã hóa bằng MD5. Tất cả các định dạng lưu trữ này được coi là không an toàn. Thay vào đó ta nên lưu trữ thông tin đăng nhập bằng cách sử dụng hàm băm mật khẩu thích ứng một chiều (tức là Bcrypt, PBKDF2, Scrypt, …) không được Digest Authentication hỗ trợ.

Digest Authentication cố gắng giải quyết nhiều điểm yếu của Basic Authentication, bằng việc đảm bảo thông tin xác thực không bao giờ được gửi rõ ràng dưới dạng văn bản.

Tiêu chuẩn xác thực thông báo Http điều chỉnh được xác định bởi RFC 2617, phiên bản cũ hơn theo chuẩn RFC2069. Hầu hết tác nhân người dùng triển khai RFC 2617.

Digest Authentication của Spring Security tương thích với chất lượng bảo vệ “auth” (qop) do RFC 2617 quy định, cung tương thích ngược với RFC 2069.

Digest Authentication được xem là một lựa chọn hấp dẫn hơn nếu ta cần sử dụng HTTP khong được mã hóa (tức là không có TLS / HTTPS) và muốn tối đa hóa tính bảo mật của của quá trình xác thực. Tuy nhiên, ta nên sử dụng HTTPS.

Trung tâm của Digest Authentication là một “nonce”. Đây là một giá trị mà server tạo ra. Spring Security’s nonce sẽ được tạo ra thông qua định dạng sau:

base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime: The date and time when the nonce expires, expressed in milliseconds

key: A private key to prevent modification of the nonce token

Ta sẽ cần phải đảm bảo rằng đa định cấu hình lưu trữ mật khẩu bằng văn bản thuần túy bằng cách sử dụng `NoOpPasswordEncoder`. Ví dụ sau cung cấp cách cấu hình Digest Authentication với Java:

@Autowired

UserDetailsService userDetailsService;

DigestAuthenticationEntryPoint entryPoint() {

DigestAuthenticationEntryPoint result = new DigestAuthenticationEntryPoint();

result.setRealmName (“My App Relam”);

result.setKey(“3028472b-da34-4501-bfd8-a355c42bdf92”);

}

DigestAuthenticationFilter digestAuthenticationFilter () {

DigestAuthenticationFilter result = new DigestAuthenticationFilter();

result.setUserDetailsService(userDetailsService);

result.setAuthenticationEntryPoint(entryPoint());

}

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) throws Exception {

http

.exceptionHandling (e -> e.authenticationEntryPoint(authenticationEntryPoint()))

.addFilterBefore(digestFilter()) ???

}

#### Lưu trữ mật khẩu.

Mỗi cơ chế được hỗ trợ để đọc tên người dùng và mật khẩu có thể tận dụng bất ky cơ chế nào được lưu trữ:

1. Lưu trữ đơn giản với xác thực trong bộ nhớ.
2. Cơ sở dữ liệu quan hệ với xác thực JDBC.
3. Lưu trữ dữ liệu tùy chỉnh với UserDetailService.
4. Bộ nhớ LDAP với xác thực LDAP.

##### Xác thực trong bộ nhớ

Spring Security `InMemoryUserDetailsManager` triển khai `UserDetailsService` để cung cấp hỗ trợ xác thực dựa trên tên người dùng/mật khẩu được lưu trữ trong bộ nhớ.

`InMemoryUserDetailsManager` cung cấp quản lý `UserDetails` bằng cách triển khai interface `UserDetailsManager`.

`UserDetails` được xác thực dựa trên cơ chế cấu hình Spring Security ở phần ‘Tên người dùng / mật khẩu’

Trong ví dụ này, ta sử dụng Spring Boot CLI để mã hóa mật khẩu và lấy mật khẩu được mã hóa:

Ví dụ: Cấu hình Java `InMemoryUserDetailsService`

@Bean

public UserDetailsService users() {

UserDetails user = User.builder()

.username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")

.roles("USER")

.build();

UserDetails admin = User.builder()

.username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")

.roles("USER", "ADMIN")

.build();

return new InMemoryUserDetailsManager(user, admin);

}

Ở ví dụ trên, ta đã lưu trữ mật khẩu ở định dạng an toàn.

Trong ví dụ sau, ta tận dụng `User.withDefaultPasswordEncoder` để đảm bảo rằng mật khẩu được lưu trữ trong bộ nhớ được bảo vệ. Tuy nhiên, nó không bảo vệ khỏi việc mật khẩu bị đoán được bằng cách dịch ngược mã nguồn. Vì lý do này `User.withDefaultPasswordEncoder` chỉ được sử dụng để ‘thử nghiệm’, không được dùng trong môi trường sản phẩm.

@Bean

public UserDetailsService users() {

// The builder will ensure the passwords are encoded before saving in memory

UserBuilder users = User.withDefaultPasswordEncoder();

UserDetails user = users

.username("user")

.password("password")

.roles("USER")

.build();

UserDetails admin = users

.username("admin")

.password("password")

.roles("USER", "ADMIN")

.build();

return new InMemoryUserDetailsManager(user, admin);

}

##### Xác thực JDBC

Spring Security `JdbcDaoImpl` triển khai `UserDetailsService` để cung cấp hỗ trợ xác thực dựa trên tên người dùng/mật khẩu được truy xuất bằng JDBC.

`JdbcUserDetailsManager` kế thừa `JdbcDaoImpl` để cung cấp quản lý `UserDetails` thông qua interface `UserDetailsManager`.

`UserDetails` được xác thực dựa trên cơ chế cấu hình Spring Security ở phần ‘Lưu tên người dùng / mật khẩu’

Trong các phần sau, ta sẽ thảo luận:

* Lược đồ quan hệ mặc định được sử dụng để xác thực JDBC của Spring Security
* Thiết lập nguồn dữ liệu
* JdbcUserDetailsManager

**Lược đồ quan hệ mặc định**

Lược đồ người dùng

`JdbcDaoImpl` yêu cầu:

* 1 bảng chứa tài khoản, mật khẩu và trạng thái mật khẩu (bật hoặc tắt).
* 1 bảng chứa các quyền người dùng

**Lưu ý**: Lược đồ mặc định cũng có thể được hiển thị dưới dạng tài nguyên classpath được đặt tên:

* /org/springframework/security/core/userdetails/jdbc/users.ddl

Ví dụ: Lược đồ mặc định người dùng:

create table users(

username varchar\_ignorecase(50) not null primary key,

password varchar\_ignorecase(500) not null,

enabled boolean not null

);

create table authorities (

username varchar\_ignorecase(50) not null,

authority varchar\_ignorecase(50) not null,

constraint fk\_authorities\_users foreign key(username) references users(username)

);

create unique index ix\_auth\_username on authorities (username,authority);

Lược đồ nhóm

Nếu ứng dụng của ta đang tận dụng các nhóm, ta sẽ cần cung cấp lược đồ nhóm. Lược đồ nhóm mặc định sẽ như ví dụ bên dưới:

create table groups (

id bigint generated by default as identity(start with 0) primary key,

group\_name varchar\_ignorecase(50) not null

);

create table group\_authorities (

group\_id bigint not null,

authority varchar(50) not null,

constraint fk\_group\_authorities\_group foreign key(group\_id) references groups(id)

);

create table group\_members (

id bigint generated by default as identity(start with 0) primary key,

username varchar(50) not null,

group\_id bigint not null,

constraint fk\_group\_members\_group foreign key(group\_id) references groups(id)

);

Thiết lập nguồn dữ liệu

Là thiết lập kết nối database trong back-end đó.

JdbcUserDetailsManager

Ví dụ:

@Bean

UserDetailsManager users(DataSource dataSource) {

UserDetails user = User.builder()

.username("user")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")

.roles("USER")

.build();

UserDetails admin = User.builder()

.username("admin")

.password("{bcrypt}$2a$10$GRLdNijSQMUvl/au9ofL.eDwmoohzzS7.rmNSJZ.0FxO/BTk76klW")

.roles("USER", "ADMIN")

.build();

JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);

users.createUser(user);

users.createUser(admin);

return users;

}

Tôi vẫn không thể hiểu??? Tại sao nó lại đéo lấy dữ liệu từ database??? – TL: vì UserDetail là dữ liệu hiện tại của người dùng ở BE, không phải dữ liệu database đúng. Sau khi có dữ liệu của người dùng, ta sẽ gửi nó xuống database để tiếp tục xác thực. `JdbcUserDetailsManager` cung cấp đủ phương thức để ta tiến hành CRUD thông tin người dùng, nhưng không nên sử dụng nó để xác thực vì nó không an toàn. Mà nên sử dụng LDAP.

##### UserDetails

`UserDetails` được trả bởi `UserDetailsService`.

`DaoAuthenticationProvider` xác thực `UserDetails` và sau đó trả về `Authentication` có giá trị là `UserDetails`, thứ được trả về chính bởi `UserDetailsService`.

##### UserDetailsService

`UserDetailsService` được `DaoAuthenticationProvider` sử dụng để lấy tên người dùng, mật khẩu và các thuộc tính khác để xác thực tên người dùng và mật khẩu.

Spring Security cung cấp các triển khai ‘xác thực trong bộ nhớ’ và ‘xác thực JDBC’ của `UserDetailsService`.

Ta có thể tự cài đặt tùy chỉnh 1 Bean `UserDetailsService` của bản thân.

**Lưu ý**: Bean tùy chỉnh `UserDetalsService` chỉ được sử dụng nếu `AuthenticationManagerBuilder` và `AuthenticationProvider` chưa được xác định.

Ví dụ:

@Bean

CustomUserDetailsService customUserDetailsService() {

return new CustomUserDetailsService();

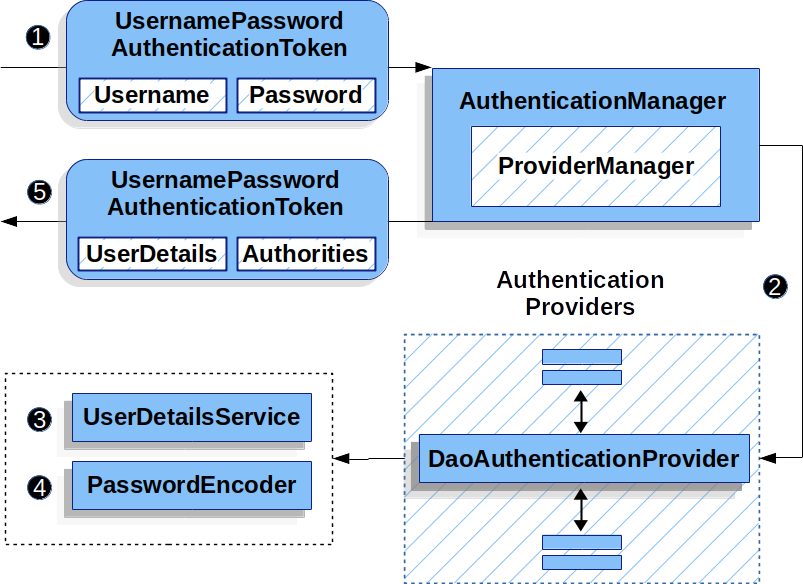
}

##### PasswordEncoder

Servlet của Spring Security hỗ trợ lưu mật khẩu một cách an toàn bằng cách tích hợp với `PasswordEncoder`. Việc tùy chỉnh `PasswordEncoder` có thể được thực hiện bằng cách khai báo một Bean `PasswordEncoder`.

##### DaoAuthenticationProvider

`DaoAuthenticationProvider` là một triển khai của `AuthenticationProvider` tận dụng `UserDetailsService` và `PasswordEncoder` để xác thực tên người dùng và mật khẩu.



1. Từ Filter dùng để đọc tên người dùng và mật khẩu chuyển đến một `UsernamePasswordAuthenticationToken` để `AuthenticationManager` thực hiện xác thực bởi `ProviderManager`
2. `ProviderManager` được cấu hình để sử dụng `DaoAuthenticationProvider` để xác thực
3. `DaoAuthenticationProvider` tra cứu `UserDetails` từ `UserDetailsService`.
4. `DaoAuthenticationProvider` sau đó sử dụng `PasswordEncoder` để xác thực mật khẩu của `UserDetails` nhận được ở bước trước.
5. Khi xác thực thành công, loại `Authentication` được trả về là `UsernamePasswordAuthenticationToken` và có mã chính là `UserDetails` được trả về bởi `UserDetailsService`. Cuối cùng, kết quả trả về `UserNamePasswordAuthenticationToken` được thiết lập `SecurityContextHolder` bởi filter xác thực.

##### Xác thực LDAP

LDAP thường được các tổ chức sử dụng như một kho lưu trữ trung tâm cho thông tin người dùng và như một dịch vụ xác thực.

Xác thực LDAP được Spring Security sử dụng, khi nó được định cấu hình để chấp nhận tên người dùng và mật khẩu để xác thực.

* Tuy nhiên, mặc dù tận dụng tên người dùng / mật khẩu để xác thực, nó lại không tích hợp `UserDetailsService`, vì trong ‘xác thực ràng buộc’ máy chủ LDAP không trả lại mật khẩu nên ứng dụng không thể thực hiện các thực mật khẩu.

Có nhiều tình huống khác nhau về cách máy chủ LDAP có thể được định cấu hình, để nhà cung cấp LDAP của Spring Security có để định cấu hình đầy đủ.

* Nó sử dụng các interface chiến lược riêng biệt để xác thực, truy xuất vai trò và cung cấp các triển khai mặc định 🡪 để xử lý nhiều tình huống.

###### Điều kiện tiên quyết

Liên kết sau, cung cấp phần giới thiệu tốt về các khái niện liên quan và hướng dẫn thiết lập thư mục bằng máy chủ LDAP miến phí OpenLDAP: [Open Source Guide - LDAP for Rocket Scientists - Contents (zytrax.com)](https://www.zytrax.com/books/ldap/). Tham khảo ApacheDS, vì LDAP server này được viết từ Java.

* Một số thông tin quen thuộc với các API JNDI được sử dụng để truy cập LDAP từ Java cũng có thể hữu ích.
* Spring Security không sử dụng bất kỳ thư viện từ bên thứ 3 nào (Mozilla, JLDAP,..) trong nhà cung cấp LDAP, nhưng được sử dụng rộng rãi là nhờ Spring LDAP, vì vậy một số thông tin quen thuộc với dữ án có thể hữu ích nếu ta có kế hoạch thêm các tùy chỉnh của riêng mình.
* Khi sử dụng xác thực LDAP, điều quan trọng là đảm bảo rằng ta định cấu hình tổng hợp kết nối LDAP đúng cách. Thảm khảo: [Connection Pooling Configuration (oracle.com)](https://docs.oracle.com/javase/jndi/tutorial/ldap/connect/config.html)

**Các dependency gồm:**

* org.springframework.boot:spring-starter-boot-data-ldap
* org.springframeword.security:spring-security-ldap

###### Thiết lập máy chủ LDAP được nhúng

Điều đầu tiên ta cần đảm bảo, là có máy chủ LDAP để trỏ đến.

Để đơn giản, tốt nhất ta nên bắt đầu với máy chủ LDAP được nhúng.

Spring Security hỗ trợ sử dụng:

* Embedded UnboundId Server – Máy chủ không liên kết được nhúng.
* Máy chủ ApacheDS được nhúng.

Ví dụ: ta sử dụng file tài nguyên classpath `users.ldif` để khởi tạo máy chủ LDAP được nhúng với người dùng `user` và `admin` đều có chung mật khẩu là `password`:

dn: ou=groups,dc=springframework,dc=org

objectclass: top

objectclass: organizationalUnit

ou: groups

dn: ou=people,dc=springframework,dc=org

objectclass: top

objectclass: organizationalUnit

ou: people

dn: uid=admin,ou=people,dc=springframework,dc=org

objectclass: top

objectclass: person

objectclass: organizationalPerson

objectclass: inetOrgPerson

cn: Rod Johnson

sn: Johnson

uid: admin

userPassword: password

dn: uid=user,ou=people,dc=springframework,dc=org

objectclass: top

objectclass: person

objectclass: organizationalPerson

objectclass: inetOrgPerson

cn: Dianne Emu

sn: Emu

uid: user

userPassword: password

dn: cn=user,ou=groups,dc=springframework,dc=org

objectclass: top

objectclass: groupOfNames

cn: user

uniqueMember: uid=admin,ou=people,dc=springframework,dc=org

uniqueMember: uid=user,ou=people,dc=springframework,dc=org

dn: cn=admin,ou=groups,dc=springframework,dc=org

objectclass: top

objectclass: groupOfNames

cn: admin

uniqueMember: uid=admin,ou=people,dc=springframework,dc=org

**Máy chủ không liên kết được nhúng -** Embedded UnboundId Server

Thêm dependency:

<dependency>

<groupId>com.unboundid</groupId>

<artifactId>unboundid-ldapsdk</artifactId>

<version>4.0.14</version>

<scope>runtime</scope>

</dependency>

Sau đó, ta có thể định cấu hình máy chủ LDAP được nhúng bằng cách sử dụng `EmbeddedLdapServerContextSourceFactoryBean`. Thao tác này sẽ hướng dẫn Spring Security khởi động máy chủ LDAP trong bộ nhớ.

Ví dụ:

@Bean

public EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean() {

return EmbeddedLdapServerContextSourceFactoryBean.fromEmbeddedLdapServer();

}

Ngoài ra, ta có thể định cấu hình Máy chủ LDAP nhúng theo cách thủ công. Nếu ta chọn cách tiếp cận này, ta sẽ phải chịu trách nhiệm quản lý vòng đời của Máy chủ LDAP được nhúng:

@Bean

UnboundIdContainer ldapContainer() {

return new UnboundIdContainer("dc=springframework,dc=org",

"classpath:users.ldif");

}

**Máy chủ ApacheDS được nhúng**

**Lưu ý**: Spring Security sử dụng Apache 1.x, thứ không còn được duy trì và cập nhật.

* Còn ApacherDS 2.x chỉ phát hành những phiên bản quan trọng mà không có bản ổn định 🡪 vì vậy, Spring Security không hỗ trợ.

Nếu ta muốn sử dụng ApacheDS, hãy thêm các phụ thuộc sau:

<dependency>

<groupId>org.apache.directory.server</groupId>

<artifactId>apacheds-core</artifactId>

<version>1.5.5</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.apache.directory.server</groupId>

<artifactId>apacheds-server-jndi</artifactId>

<version>1.5.5</version>

<scope>runtime</scope>

</dependency>

Sau đó, ta có thể định cấu hình máy chủ LDAP được nhúng:

@Bean

ApacheDSContainer ldapContainer() {

return new ApacheDSContainer("dc=springframework,dc=org",

"classpath:users.ldif");

}

###### LDAP ContextSource

Khi ta đã có máy chủ LDAP để trỏ đến, ta cần định cấu hình Spring Security để trỏ đến máy chủ LDAP sẽ được sử dụng để xác thực người dùng.

* Điều này sẽ được thực hiện bằng cách tạo LDAP `ContextSource`, tương tự với JDBC `DataSource`.

Nếu ta đã định cấu hình một `EmbeddedLdapServerContextSourceFactoryBean`, Spring Security sẽ tạo một LDAP `ContextSource` trỏ đến máy chủ LDAP được nhúng.

@Bean

public EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean() {

EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean =

EmbeddedLdapServerContextSourceFactoryBean.fromEmbeddedLdapServer();

contextSourceFactoryBean.setPort(0);

return contextSourceFactoryBean;

}

Ngoài ra, ta có thể định cấu hình rõ ràng `ContextSource` để kết nối đến máy chủ LDAP được cung cấp – không phải máy chủ nhúng nhé

ContextSource contextSource(UnboundIdContainer container) {

return new DefaultSpringSecurityContextSource("ldap://localhost:53389/dc=springframework,dc=org");

}

###### Xác thực

Hỗ trợ LDAP của Spring Security không sử dụng `UserDetailsService`, vì xác thực liên kết LDAP không cho phép máy khách đọc mật khẩu hoặc thậm chí là phiên bản băm của mật khẩu.

* Điều này có nghĩa là không có cách nào một mật khẩu được đọc và sau đó được xác thực bởi Spring Security.

Vì lý do này, LDAP được hỗ trợ triển khai bằng interface `LdapAuthenticator`. `LdapAuthenticator` cũng chịu trách nhiệm truy xuất bất kỳ thuộc tính người dùng cần thiết nào.

* Điều này là do quyền đối với các thuộc tính có thể phụ thuộc vào các loại xác thực đang được sử dụng.
* Ví dụ: Nếu ràng buộc với tư cách là người dùng, có thể cần phải đọc chúng với các quyền riêng của người dùng.

Có 2 cách để triển khai `LdapAuthenticator`:

* Xác thực ràng buộc.
* Xác thực mật khẩu.

###### Xác thực ràng buộc với ldap nhúng

Xác thực ràng buộc là cơ chế phổ biến nhất để xác thực người dùng bằng LDAP.

* Trong xác thực ràng buộc, thông tin xác thực của người dùng (tức là tên người dùng/mật khẩu) được gửi đến máy chủ LDAP để xác thực.
* Ưu điểm của việc sử dụng xác thực ràng buộc là các bí mật của người dùng (tức mật khẩu) không cần phải tiết lộ cho máy khách 🡪 giúp bảo vệ khỏi rò rỉ.

Ví dụ: Áp dụng với các Ldap được nhúng.

@Bean

AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource) {

LdapBindAuthenticationManagerFactory factory = new LdapBindAuthenticationManagerFactory(contextSource);

factory.setUserDnPatterns("uid={0},ou=people");

return factory.createAuthenticationManager();

}

Ví dụ trên, sẽ lấy **DN** của người dùng bằng cách **thay thế** tên đăng nhập của người dùng trong form login và cố giắng liên kết với tư cách người dùng đó bằng mật khẩu được cung cấp.

* Điều này là OK nếu tất cả người được lưu trữ dưới 1 nút duy nhất trong thư mục.

Nếu ta muốn cấu hình bộ lọc tìm kiếm LDAP để xác định vị trí người dùng, ta có thể sử dụng như sau – Áp dụng với Ldap được nhúng.

@Bean

AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource) {

LdapBindAuthenticationManagerFactory factory = new LdapBindAuthenticationManagerFactory(contextSource);

factory.setUserSearchFilter("(uid={0})");

factory.setUserSearchBase("ou=people");

return factory.createAuthenticationManager();

}

Nếu được sử dụng với ContextSource, điều này sẽ thực hiện tìm trong DN: ou=people,dc=springframework,dc=org bằng cách sử dụng (uid={0}) như một bộ lọc. Một lần nữa, tên đăng nhập của người dùng được thay thế cho tên tham số trong bộ lọc, vì vậy nó sẽ tìm kiếm mục nhập có thuộc tính **uid** bằng với tên người dùng. Nếu cơ sở tìm kiếm của người dùng không được cung cấp, việc tìm kiếm sẽ được thực hiện từ gốc.

###### Xác thực ràng buộc với Ldap Server

**Ví dụ**:

@Configuration

@RequiredArgsConstructor

public class SecurityConfig {

private final BaseLdapPathContextSource contextSource;

@Bean

public BindAuthenticatior ldapAuthenticator () {

BindAuthenticator authenticator = new BindAuthenticator (contextSource);

authenticator.setUserSearch (new FilterBasedLdapUserSearch (“ou=people”, “uid={0}”, contextSource));

return authenticator;

}

@Bean

public LdapAuthenticatonProvider ldapAuthenticationProvider () {

return new LdapAuthenticationProvider (ldapAuthenticator());

}

}

**Lưu ý**: Bean BaseLdapPathContextSource có thể được tạo thành bằng cách khai báo trong file ‘application.properties’ như sau:

spring.ldap.base=<root>

spring.ldap.urls[0]=ldap://127.0.0.1:389 #389 là port mặc định

spring.ldap.username=<admin>

spring.ldap.password=<mật khẩu của admin>

**Lưu ý**:

* Phương pháp xác thực ràng buộc sẽ gửi thông tin của người dùng gồm: username, mật khẩu, … chưa mã hóa xuống ldap, ldap sẽ mã hóa mật khẩu và so khớp sau đó phản hồi lại về server.

###### Xác thực mật khẩu với LDAP nhúng

Xác thực mật khẩu là khi mật khẩu do người dùng cung cấp được so sánh với mật khẩu được lưu trữ trong kho lưu trữ.

* Điều này có thể được thực hiện bằng cách truy xuất giá trị của thuộc tính mật khẩu và kiểm tra nó cục bộ bằng cách thực hiện thao tác “so sánh” LDAP
* Trong đó, mật khẩu đã cung cấp được băm và chuyển đến LDAP để so sánh và giá trị mật khẩu thực không bao giờ được truy xuất.
* Không thể thực hiện so sánh LDAP khi mật khẩu được băm với một <salt> ngẫu nhiên.

Ví dụ: Cấu hình xác thực mật khẩu tối thiểu

@Bean

AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource) {

LdapPasswordComparisonAuthenticationManagerFactory factory = new LdapPasswordComparisonAuthenticationManagerFactory(

contextSource, NoOpPasswordEncoder.getInstance());

factory.setUserDnPatterns("uid={0},ou=people");

return factory.createAuthenticationManager();

}

Ta cũng có thể cấu hình nâng cao hơn như sau:

@Bean

AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource) {

LdapPasswordComparisonAuthenticationManagerFactory factory = new LdapPasswordComparisonAuthenticationManagerFactory(

contextSource, new BCryptPasswordEncoder());

factory.setUserDnPatterns("uid={0},ou=people");

factory.setPasswordAttribute("pwd");

return factory.createAuthenticationManager();

}

###### Xác thực mật khẩu với LDAP Server.

**Ví dụ**:

@Configuration

@RequiredArgsConstructor

public class SecurityConfig extends WebSecurityConfigurerApdater {

private final BaseLdapPathContextSource contextSource;

@Override

public void configure (AuthenticationManagerBuilder auth) throws Exception {

auth.ldapAuthentication()

.userDnPatterns (“uid={0},ou=people”)

.contextSource(contextSoure)

.passwordCompare ()

.passwordEncoder (new LdapShaPasswordEncoder())

.passwordAttribute (“userPassword”)

}

}

**Lưu ý**:

1. Không nên dùng xác thực mật khẩu, Ldap không hỗ trợ các kiểu mã hóa hiên tại của Spring Security và `LdapShaPasswordEncoder` đã lỗi thời và không đủ an toàn.
2. Phương pháp xác nhận mật khẩu này sẽ nhận mật khẩu chưa mã hóa của người dùng, sau đó mã hóa và gửi về Ldap để so khớp.

###### LdapAuthoritiesPopulator

**`**LdapAuthoritiesPopulator` được sử dụng để xác định quyền hạn nào được trả về cho người dùng.

@Bean

LdapAuthoritiesPopulator authorities(BaseLdapPathContextSource contextSource) {

String groupSearchBase = "";

DefaultLdapAuthoritiesPopulator authorities =

new DefaultLdapAuthoritiesPopulator(contextSource, groupSearchBase);

authorities.setGroupSearchFilter("member={0}");

return authorities;

}

@Bean

AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource, LdapAuthoritiesPopulator authorities) {

LdapBindAuthenticationManagerFactory factory = new LdapBindAuthenticationManagerFactory(contextSource);

factory.setUserDnPatterns("uid={0},ou=people");

factory.setLdapAuthoritiesPopulator(authorities);

return factory.createAuthenticationManager();

###### Thư mục hoạt động.

**Lưu ý**: đại loại đây là 1 cách kết nối LDAP Server.

Thư mục hoạt động hỗ trợ các tùy chọn xác thực của riêng nó không theo bất kỳ tiêu chuẩn nào và kiểu sử dụng thông thường không quá phù hợp với tiêu chuẩn `LdapAuthenticationProvider`.

* Thông thường, xác thực được thực hiện bằng tên người dùng miền (trong biểu mẫu user@domain), thay vì sử dụng tên phân biệt LDAP
* để làm cho điều này dễ dàng hơn, Spring Security có một nhà cung cấp xác thực được tùy chỉnh cho thiết lập ‘Thư mục hoạt động’ điển hình.

Cấu hình `ActiveDirectoryLdapAuthenticationProvider` khá đơn giản. Ta chỉ cần cung cấp tên miền và url ldap địa chỉ máy chủ.

Ví dụ:

@Bean

ActiveDirectoryLdapAuthenticationProvider authenticationProvider() {

return new ActiveDirectoryLdapAuthenticationProvider("example.com", "ldap://company.example.com/");

}

## Xác thực liên tục

Vào lần đầu tiên người dùng truy cập hệ thống và yêu cầu các tài nguyên đòi hỏi xác thực, server sẽ yêu cầu người dùng nhập “thông tin đăng nhập“. Một trong những cách phổ biến nhất để yêu cầu người dùng đăng nhập, đó là chuyển hướng đến trang đăng nhập.

**Ví dụ**: Cách người dùng được xác thực mật khẩu:

GET / HTTP/1.1

Host: example.com

Cookie: SESSION=91470ce0-3f3c-455b-b7ad-079b02290f7b

HTTP/1.1 302 Found

Location: /login

Người dùng gửi tên người dùng và mật khẩu:

POST /login HTTP/1.1

Host: example.com

Cookie: SESSION=91470ce0-3f3c-455b-b7ad-079b02290f7b

username=user&password=password&\_csrf=35942e65-a172-4cd4-a1d4-d16a51147b3e

Sau khi xác thực thành công, người dùng được liên kết với 1 session (đồng thời tạo cookie bên phía người dùng) mới:

HTTP/1.1 302 Found

Location: /

Set-Cookie: SESSION=4c66e474-3f5a-43ed-8e48-cc1d8cb1d1c8; Path=/; HttpOnly; SameSite=Lax

Các yêu cầu tiếp theo sẽ được xác thực thông qua cookie từ phía người dùng:

GET / HTTP/1.1

Host: example.com

Cookie: SESSION=4c66e474-3f5a-43ed-8e48-cc1d8cb1d1c8

### Kho lưu trữ các thông tin bảo mật người dùng

Trong Spring Security, việc liên kết người dùng với các request trong tương lai được thực hiện bằng cách sử dụng `SecurityContextRepository`.

#### HttpSessionSecurityContextRepositoy

Triển khai mặc định của `SecurityContextRepository` là `HttpSessionSecurityContextRepository`, nó có tác dụng liên kết `SecurityContext` với `HttpSession`.

#### NullSecurityContextRepository

Nếu ta không muốn liên kết `SecurityContext` với `HttpSession` (như khi xác thực bằng OAuth) thì ta nên sử dụng `NullSecurityContextRepository`.

#### RequestAttibuteSecurityContextRepository

` RequestAttibuteSecurityContextRepository` sẽ tiến hành lưu thuộc tính `SecurityContext` trong phạm vi request để đảm bảo `SecurityContext` này khả dụng trong một yêu cầu duy nhất.

**Ví dụ**: Sử dụng RequestAttributeSecurityContextRepository

public SecurityFilterChain filterChain (HttpSecurity http) {

http

.securityContext ( (securityContext) -> securityContext

.securityContextRepository (new RequestAttributeSecurityContextRepository() ) );

return http.build();

}

### SecurityContextPersistenceFilter

`SecurityContextPersistenceFilter` chịu trách nhiệm **duy trì** SecurityContext giữa các request bằng cách sử dụng `SecurityContextRepository`.



1. Trước khi chạy phần còn lại của ứng dụng, SecurityContextPersistenceFilter sẽ tải SecurityContext từ SecurityContextRepository. Sau đó lưu SecurityContext vào SecurityContextHolder.
2. Tiếp theo, ứng dụng được chạy.
3. Cuối cùng, nếu SecurityContext thay đổi, ta sẽ lưu Securitycontext bằng cách sử dụng SecurityContextPersistenceFilter. Ý nghĩa ở đây khi sử dụng SecurityContextPersistenceFilter là: ta chỉ cần cài đặt SecurityContextHolder, còn SecurityContext sẽ luôn được duy trì khi đang sử dụng SecurityRepository.

Trong một số trường hợp, một response được commit và ghi vào client trước khi phương thức SecurityContextPersistenceFilter hoàn thành.

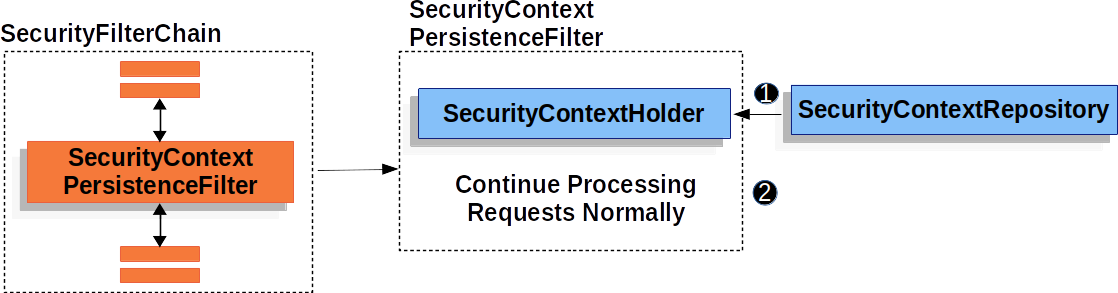
**Ví dụ**: Nếu một yêu cầu của người dùng bị chuyển hướng, response sẽ ngay lập tức bị ghi lại cho client. Điều này có nghĩa là việc thiết lập HttpSession tại bước 3 sẽ không thể thành công vì sessionId sẽ không được đưa vào reponse đã được viết.

**Ví dụ**: Một tình huống khác có thể xảy ra là nếu client xác thực thành công, nhưng phản hồi bị commit trước khi SecurityContextPersistenceFilter hoàn tất. Trong thời gian đó người dùng lại gửi 1 request thứ 2, trong khi SecurityContextPersistenceFilter trước đó kịp hoàn thành 🡪 xác sai có thể xuất hiện tại request thứ 2.

🡺Để tránh các điều trên, SecurityContextPersistenceFilter phải chứa cả HttpServletRequest và HttpServletResponse để phát hiện SecurityContext có thay đổi hay không? Nếu có, tiến hành lưu SecurityContext trước khi phản hồi lại người dùng.

### SecurityContextHolderFilter

SecurityContextHolderFilter chịu trách nhiệm **tải** SecurityContext giữa các request bằng cách sử dụng SecurityContextHolderFilter.



1. Trước khi chạy phần còn lại của ứng dụng, SecurityContextHolderFilter tải SecurityContext từ SecurityContextRepository. Sau đó gán SecurityContext đó vào SecurityContextHolder.
2. Tiếp tục chạy ứng dụng.

Không giống như SecurityContextPersistenceFilter, SecurityContextHolderFilter sẽ tiến hành tải SecurityContext với mỗi request.

**Ví dụ**: sử dụng SecurityContextHolderFilter

public SecurityFilterChain filterChain (HttpSecurity http) {  
 http

.securityContext ((securityContext) 🡪 securityContext.requiredExplicitSave(true));

return http.build();

}

## Quản lý Session

### Bắt buộc tạo Session

Được thực hiện bằng cách sử dụng ForceEagerSessionCreationFilter.

**Ví dụ**:

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) {

http

.sessionManagement (session ->

session.sessionCreationPolicy (SessionPolicy.ALWAYS) );

return http.build();

}

### Quy định thời gian chờ

Ta có thể định cấu hình Spring Security để quy định sau bao nhiêu lâu sessionId không còn hợp lệ và chuyển hướng người dùng đến một url thích hợp. Điều này đạt được thông qua `session-management`:

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) {  
 http

.sessionManagement (session -> session.invalidSessionUrl(“/invalidSession.html”));

return http.build();

}

**Lưu ý**: Nếu ta sử dụng cơ chế này để phát hiện thời gian chờ của session, nó có thể báo lỗi sai nếu người dùng đăng xuất rồi đăng nhập lại mà không đóng trình duyệt. Điều này là do cookie của session không bị xóa khi ta vô hiệu hóa session và sẽ được gửi lại ngay cả khi người dùng đã đăng xuất.

**Ví dụ**: Ta có thể xóa rõ ràng cookie JSESSIONID khi đăng xuất bằng cách như sau:

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) {

http

.logout (logout -> logout.deleteCookies (“JSESSIONID”));

return http.build();

}

**Lưu ý**: Thật không may, điều này không thể được đảm bảo để hoạt động với mọi bộ chứa Servlet, vì vậy ta cần kiểm tra nó trong môi trường của mình.

Nếu ta đang chạy ứng dụng của mình sau proxy, ta cũng có thể xóa cookie của session bằng cách định cấu hình máy chủ proxy.

**Ví dụ**: Sử dụng `mod\_headers` của Apache HTTPD, lệnh sau sẽ xóa cookie JSESSIONID khi người dùng đăng xuất (giả sử dụng được triển khai theo đường dẫn /tutorial) :

<LocationMatch "/tutorial/logout">

Header always set Set-Cookie "JSESSIONID=;Path=/tutorial;Expires=Thu, 01 Jan 1970 00:00:00 GMT"

</LocationMatch>

### Kiểm soát các Session đăng nhập của người dùng

Nếu ta muốn đặt các ràng buộc đối với khả năng đăng nhập vào ứng dụng của người dùng. Spring Security có hỗ trợ tính huống này đơn gian như sau:

Trước tiên, ta cần thêm Bean sau để lắng nghe các sự kiện về vòng đời của session:

@Bean

public HttpSessionEventPublisher httpSessionEventPublisher() {

return new HttpSessionEventPublisher();

}

Sau đó thêm các dòng sau vào ngữ cảnh của ứng dụng:

@Bean

public SecurityFilterChain filterChain (HttpSecurity http) {  
 http

.sessionManagement (session -> session.maximumSessions(1));

return http.build();

}

Điều này sẽ ngăn người dùng đăng nhập nhiều lần – lần đăng nhập thứ 2 sẽ khiến lần đăng nhập thứ nhất bị vô hiệu.

Nếu ta muốn ngăn lần đăng nhập thứ 2, trong trường hợp đó ta có thể sử dụng:

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) {

http

.sessionManagement(session -> session

.maximumSessions(1)

.maxSessionsPreventsLogin(true)

);

return http.build();

}

Lần đăng nhập thứ 2 sau đó sẽ bị từ chối. Khi đăng nhập thứ 2 bị từ chối, người dùng sẽ được chuyển đến `authentication-failure-url` của `formLogin`.

* Nếu đăng nhập lần thứ 2 diễn ra theo cơ chế không tương tác khác, chẳng hạn như “ghi nhớ tôi”, 1 lỗi “unauthorized” (401) sẽ được gửi đến client.
* Nếu như ta muốn sử dụng một trang lỗi thay vì `authentication-failure-url`, ta có thể thêm thuộc tính `session-authentication-error-url` vào phần tử `session-management`.

### Bảo vệ tấn công cố định phiên

Các cuộc tấn công cố định phiên là một nguy cơ tiềm ẩn khi kẻ tấn công độc hại có thể tạo phiên bằng cách truy cập vào một trang web, sau đó thuyết phục người dùng khác đăng nhập cùng 1 phiên với kẻ đó (ví dụ: họ sử dụng một liên kết chứa sessionId làm tham số).

Spring Security mặc định bảo vệ khỏi các cuộc tấn công kiểu này bằng cách tạo phiên mới hoặc thay đổi sessionId khi người dùng đăng nhập.

Nếu ta muốn tắt chức năng này vì nó xung đột với một số yêu cầu khác, … Ta có thể thực hiện bằng cách sử dụng thuộc tính `session-fixation-protection` trong <session-management>. Trong có có 4 lựa chọn:

1. none: không làm gì cả. Phiên ban đầu sẽ được giữ lại.
2. newSession: tạo một phiên mới, không sao chép dữ liệu của phiên hiện có (nhưng các thuộc tính liên qua đến Spring Security vẫn sẽ được sao chép).
3. migrateSession: Tạo một phiên mới và sao chép tất cả thuộc tính phiên hiện có sang phiên mới. Đây là giá trị mặc định trong các bộ chứa Servlet 3.0 trở xuống.
4. changeSessionId: Không tạo phiên làm việc mới. Thay vào đó, hãy sử dụng cách bảo vệ phiên khỏi các cuộc tấn công cố định phiên được cung cấp bởi bộ chứa Servlet (HttpServletRequest#changeSessionId()). Tùy chọn này chỉ khả dụng trong Servlet 3.1 (Java EE 7) và các vùng chứa mới hơn. Việc chỉ định nó trong các vùng chứa cũ hơn sẽ dẫn đến ngoại lệ. Đây là giá trị mặc định trong Servlet 3.1 và các vùng chúa mới hơn.

**Lưu ý**: Khi Spring Security bảo vệ ứng dụng khỏi các cuộc tấn công cố định phiên, nó sẽ ném ra event SessionFixationProtectionEvent và bị bắt bởi javax.servlet.http.HttpSessionIdListener. Và nếu ta sử dụng cơ chế changeSessionId, biện pháp này cũng bị bắt bởi javax.servlet.http.HttpSessionIdListener. Vì vậy hãy cẩn trọng nếu mã của ta lắng nghe cả 2 sự kiện.

### Bộ lọc quản lý phiên

`SessionManagementFilter` sẽ kiểm tra nội dung của `SpringContextRepository` bằng cách đối chiếu nỗi dụng của nó với `SecurityContextHolder` nhằm xác định xem người dùng đã được xác thực trong yêu câu hiện tại hay chưa (được áp dụng với các yêu cầu xác thực không tương tác, như: người dùng đã đăng nhập trước đó rồi, hoặc remember-me). Nếu repository chứa một security context, the filter sẽ không làm gì. Nếu repository không chứa security context và `SecurityContext` chứa một đối tượng `Authentication`, thì filter sẽ giả định là người dùng đã xác thực trước đó và gọi `SessionAuthenticationStrategy`.

Nếu người dùng hiện chưa được xác thực, bộ lọc sẽ kiểm tra xem sessionId không hợp lệ đã được yêu cầu hay chưa (ví dụ: do hết thời gian chờ) và sẽ gọi câu hình `InvalidSessionStrategy`. Hành vi phổ biến nhất là chỉ chuyển hướng đến 1 Url cố định.

Các phần tiếp theo bao giờ cần thì học.

## OAuth 2.0

### Đăng nhập OAuth 2.0

Tính năng đăng nhập OAuth 2.0 cung cấp cho ứng dụng khả năng để người dùng xác thực bằng cách sử dụng tính năng xác thực của nhà cung cấp OAuth 2.0 (ví dụ: Github) hoặc nhà cung cấp OpenId Connect 1.0 (ví dụ: Google).

### Cấu hình cơ bản

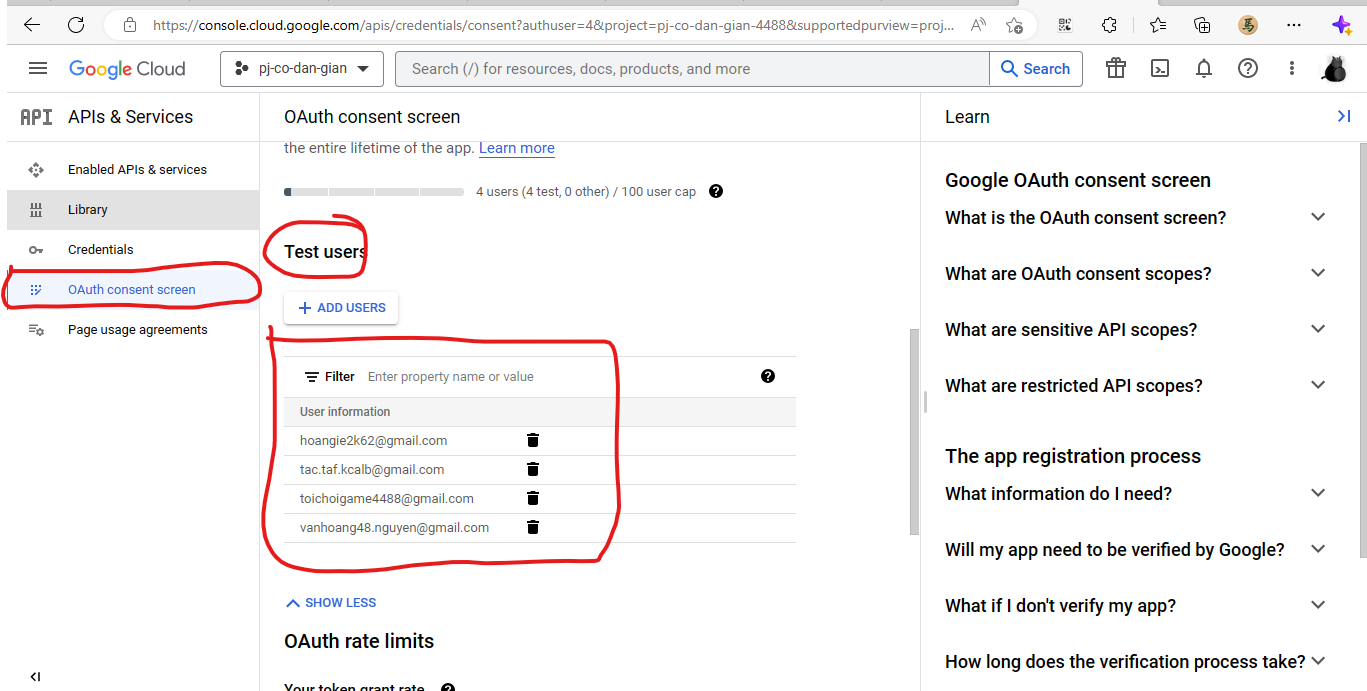
Phần dưới đây sẽ trình bầy về cách định cấu hình đăng nhập OAuth 2.0 bằng Google.

#### Thiết lập ban đầu

**Dependency**: org.springframework.boot:spring-boot-starter-oauth2-client

Trước tiên, ta cần đăng ký “tính năng xác thực” của Google bằng Google Api Console : <https://developers.google.com/identity/openid-connect/openid-connect>

**Lưu ý**: Việc sử dụng Google Api Console để xác thực thông tin người dùng, đòi hỏi nhà phát triển và ứng dụng được google xác thực. Vì vậy, khi ứng dụng vẫn còn trong giai đoạn phát triển, nhà phát triển phải đăng ký rõ ràng các gmail user sẽ test sản phẩm.



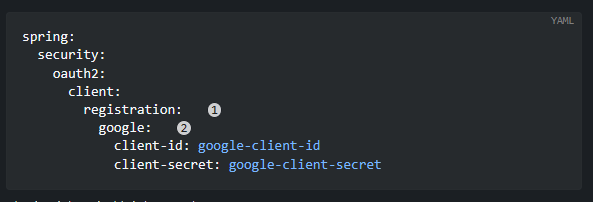
#### Đặt redirect uri

Redirect Uri là đường dẫn sau khi người dùng đã xác thực thành công với google.

**Ghi nhớ**: mẫu url mặc định là: {baseUrl}/login/oauth2/code/{registrationId}:

* baseUrl là domain và port của ứng dụng.
* registrationId là mã định danh duy nhất cho ClientRegistration

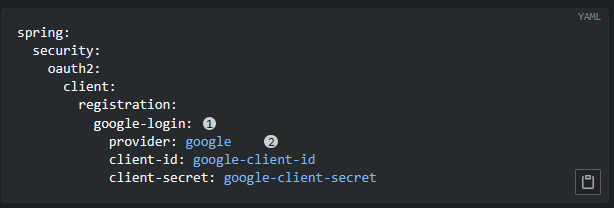
#### Định cấu hình application.yml



**Lưu ý**: Việc cấu hình bằng file application.yml 🡪 ta không cần cấu hình bằng bean trong file @Configuration.

* registrationId google ở đây khớp với enum GOOGLE (không phân biệt hoa thường) trong `CommonOAuth2Provider`.

Nếu ta muốn đặt 1 registrationId của riêng ta, ta làm như sau:



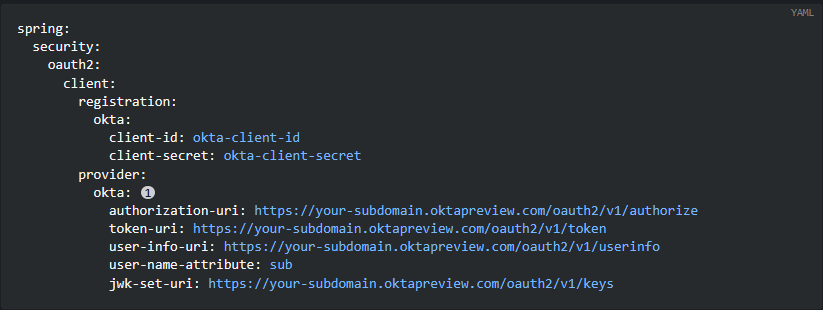
Tham khảo các thuộc tính tại đường link: [Core Configuration :: Spring Security](https://docs.spring.io/spring-security/reference/5.7.7/servlet/oauth2/login/core.html)

#### Nhà cung cấp CommonOAuth2

`CommonOAuth2Provider` xác định trước 1 tập hợp các thuộc tính cho các 1 số nhà cung cấp nổi tiếng như: Google, Github, Facebook và Okta.

Vì `authorization-uri`, `token-uri` và `user-info-uri` của các các nhà cung cấp không thay đổi thường xuyên. Dó đó các giá trị này được đặt mặc định để giám câu hình cần thiết.

#### Định cấu hình nhà cung cấp tùy chỉnh



#### Ghi đè cấu hình tự động Spring Boot 2.x

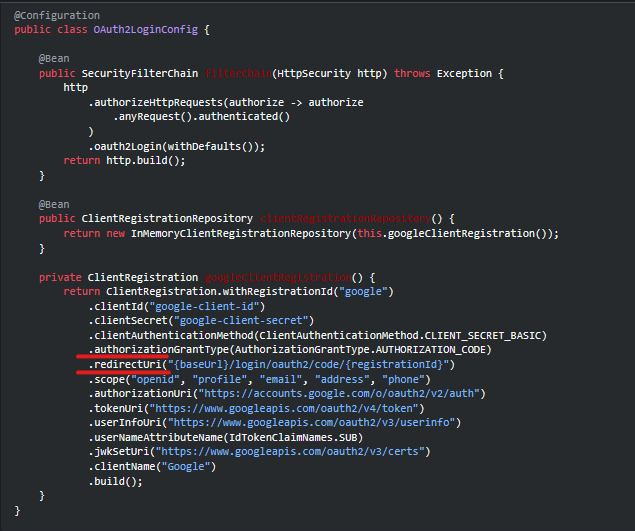
Lớp cấu hình tự động Spring Boot 2.x để hỗ trợ máy khác OAuth là `OAuth2ClientAutoConfiguration`.

Nó thực hiện các nhiệm vụ sau:

* Đăng ký Bean ClientRegistrationRepository chứa ClientRegistration gồm các thuộc tính máy khách OAuth đã định cấu hình.
* Đăng ký Bean SecurityFilterChain và bật đăng nhập OAuth 2.0 thông qua httpSecurity.oauth2Login().

#### Đăng ký Bean ClientRegistrationRepository

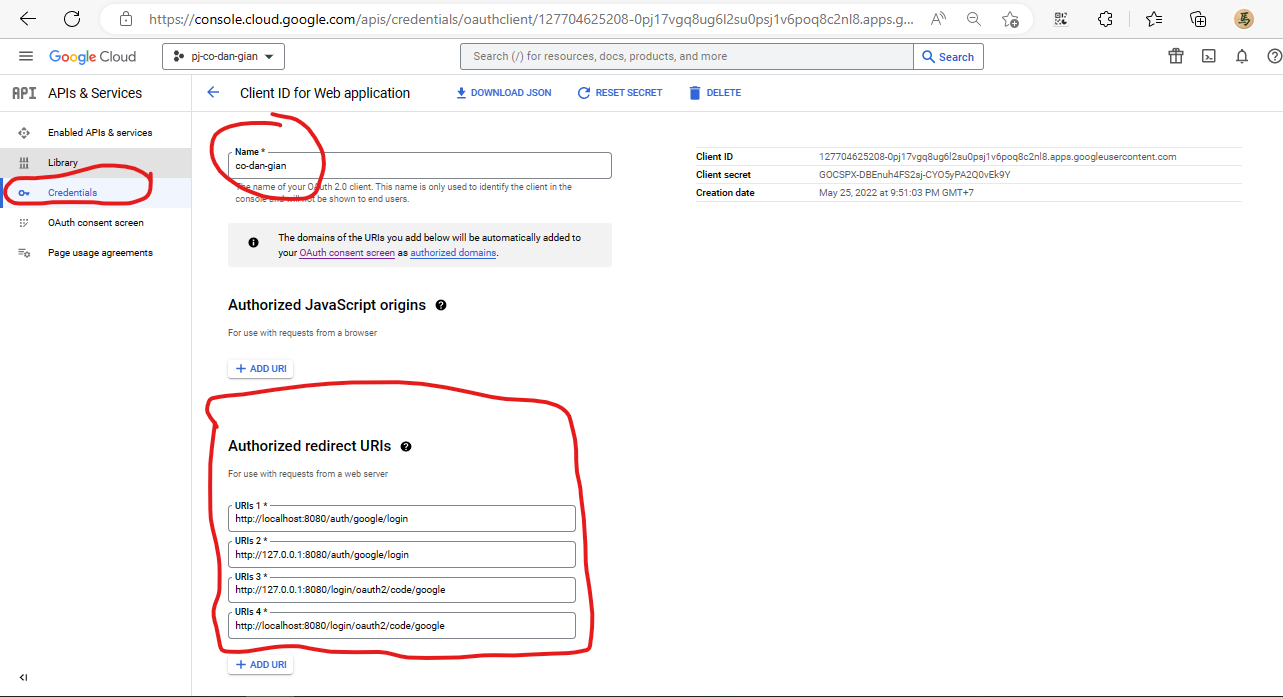
**Ví dụ**:



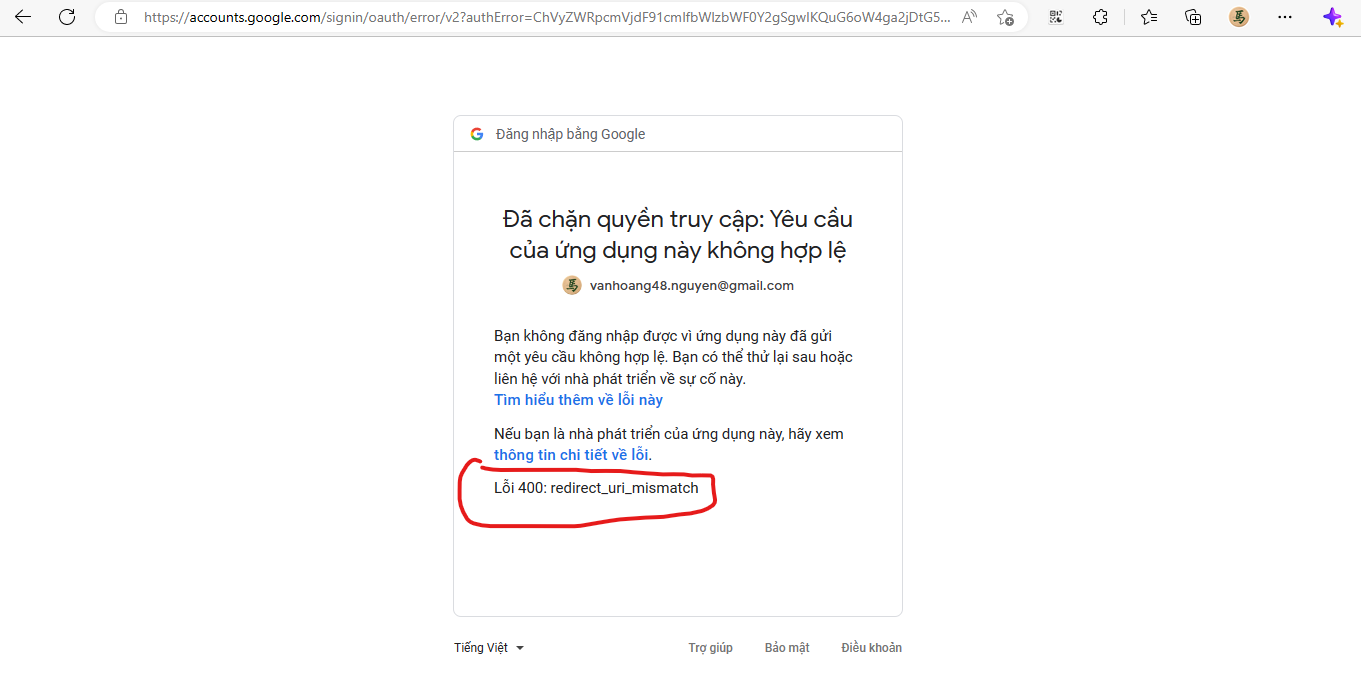
việc khai báo ‘withDefaults()’ hay public void customize(OAuth2LoginConfigurer<HttpSecurity> httpSecurityOAuth2LoginConfigurer) không làm gì như thế này là để cấu hình ‘oauth2Login’ dựa theo Bean

**Ghi nhớ**:

1. Thứ tự oauth2Login() và formLogin() không quan trọng, cái nào trước, cài nao sau đều không ảnh hưởng đến chương trình.
2. redirectUri sẽ có dạng như sau: <http://localhost:8080/auth/google/login>. Để google chấp nhận redirectUri làm địa chỉ gửi phản hồi sau khi xác thực thành công, ta cần đăng ký nó trong google api console như sau (redirect uri phải là giao thức GET và phải được cấu hình với Spring Security là không cần xác thực – nếu không sau khi xác thực thành công với google, sẽ không chuyển tiếp đến uri khai báo):



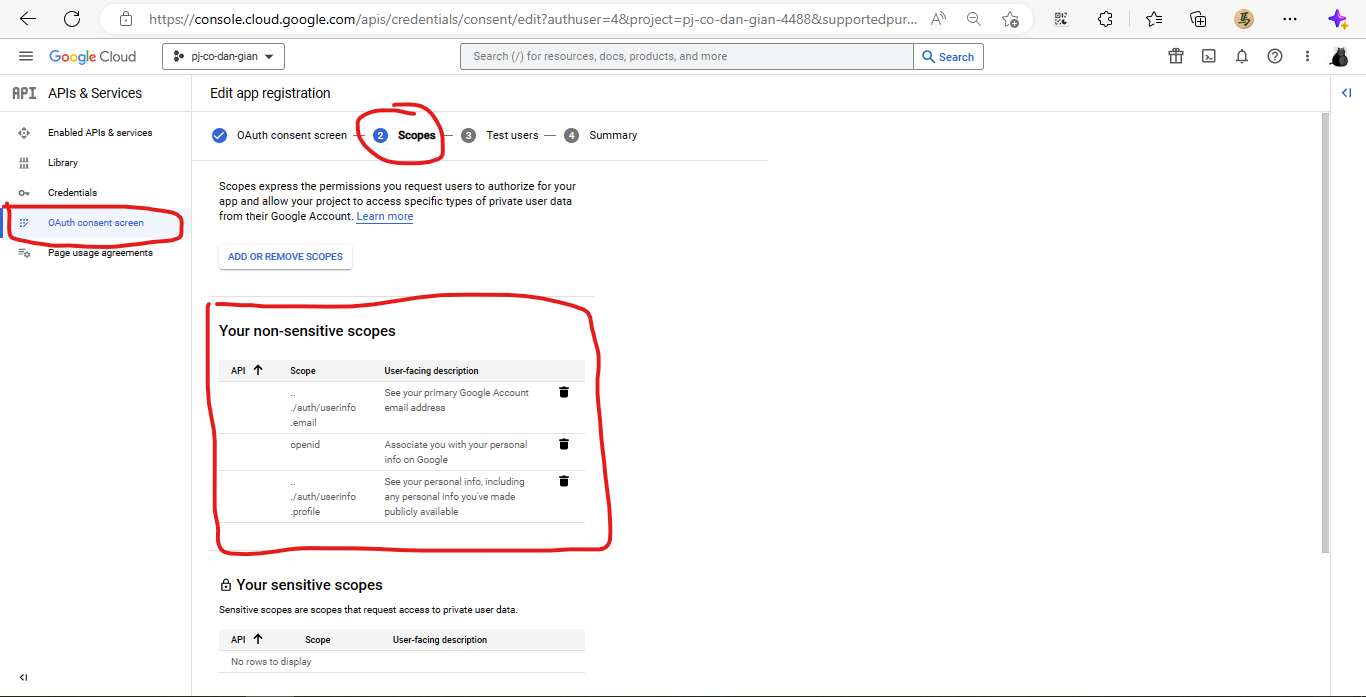
Nếu như ta không đăng ký redirectUri với google, thì khi người dùng sử dụng chức năng đăng nhập bằng google, sẽ nhận thông báo như sau:



1. scope khai báo trong ClientRegistration phải nhỏ hơn hoặc bằng phạm vi yêu cầu của ứng dụng khai bào trên Google Api Console, nếu không sẽ nhận phản hồi:

[invalid\_token\_response] An error occurred while attempting to retrieve the OAuth 2.0 Access Token Response: 401 Unauthorized: [no body]

Hình sau là nơi khai báo phạm vi ứng dụng:



# Ứng dụng Reactive

Các ứng dụng reactive hoạt động rất khác so với ứng dụng servlet. Phần này thảo luận về cách Spring Security hoạt động với các ứng dụng reactive được **viết** bằng cách sử dụng Spring WebFlux.

## Bắt đầu

**dependecy**:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

## Xác thực - Authentication

### Xác thực X.509

Tương tự như xác thực Servlet X.509, bộ lọc xác thực x509 reactive cho phép trích xuất access\_token từ chứng chỉ do client cung cấp.

Ví dụ về cấu hình bảo mật x509 reactive:

@Bean

public SecurityWebFilterChain securityWebFilterChain (ServerHttpSecurity http){

http.x509(withDefaults())

.authorizeExchange(exchange 🡪 exchange.anyChange().permitAll());

return http.build();

}

Trong cấu hình trên, khi không cung cấp **principalExtractor** hoặc **authenticationManager** 🡪 giá trị mặc định sẽ được sử dụng.

Trình trích xuất (extractor) mặc định là SubjectDnX509PrincipalExtractor, công cụ trích xuất trường **CN** (common name) từ chứng chỉ do client cung cấp.

Trình quản lý (manager) mặc định là ReactivePreAuthenticateAuthenticationManager. Đây là công cụ thực hiện xác thực tài khoản người dùng, kiểm tra xem tài khoản người dùng bị trích xuất (extractor) có tồn tại principalExtractor hay không? Có bị khóa, vô hiệu hóa hoặc hết hạn hay không?

Ví dụ minh họa cách ghi đè các giá trị mặc định này:

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

SubjectDnX509PrincipalExtractor principalExtractor =

new SubjectDnX509PrincipalExtractor();

principalExtractor.setSubjectDnRegex("OU=(.\*?)(?:,|$)");

ReactiveAuthenticationManager authenticationManager = authentication -> {

authentication.setAuthenticated("Trusted Org Unit".equals(authentication.getName()));

return Mono.just(authentication);

};

http

.x509(x509 -> x509

.principalExtractor(principalExtractor)

.authenticationManager(authenticationManager)

)

.authorizeExchange(exchanges -> exchanges

.anyExchange().authenticated()

);

return http.build();

}

Trong ví dụ này, tên người dùng được trích xuất từ trường OU của chứng chỉ Client thay vì CN và **việc tra cứu tài khoản bằng cách sử dụng** ReactiveUserDetailService **hoàn toàn không được thực hiện**.

Vui lòng tham khảo: [ví dụ về định cấu hình Netty và WebClient hoặc công cụ dòng lệnh curl để sử dụng chung TLS và bật xác thực X.509](https://github.com/spring-projects/spring-security-samples/tree/main/servlet/java-configuration/authentication/x509)

### Đăng xuất

Spring Security cung cấp một điểm cuối đăng xuất theo mặc định. Sau khi đăng nhập, ta có thể GET /logout hoặc POST /logout để đăng xuất. Khi đó:

* Xóa ServerCsrfTokenRepository và ServerSecurityContextRepository
* Chuyển hướng trở lại trang đăng nhập.

Thông thường, ta cũng sẽ muốn làm mất hiệu lực của session khi đăng xuất. Để đạt điều này, ta có thể thêm WebSessionServerLogoutHandler vào cấu hình đăng xuất của mình như sau:

@Bean

SecurityWebFilterChain http(ServerHttpSecurity http) throws Exception {

DelegatingServerLogoutHandler logoutHandler = new DelegatingServerLogoutHandler(

new WebSessionServerLogoutHandler(), new SecurityContextServerLogoutHandler()

);

http

.authorizeExchange((exchange) -> exchange.anyExchange().authenticated())

.logout((logout) -> logout.logoutHandler(logoutHandler));

return http.build();

}

## Xác quyền – Authoization

### Xác quyền request http

Spring Security cung cấp hỗ trợ xác quyền cho các request http. Theo mặc định xác quyền của Spring Security, sẽ yêu cầu tất cả các request phải được xác thực trước.

Ví dụ:

@Bean

SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {

http

.authorizeExchange(exchanges -> exchanges

.anyExchange().authenticated()

)

.httpBasic(withDefaults())

.formLogin(withDefaults());

return http.build();

}

Ta có thể định cấu hình Spring Security để có các quy tắc khác nhau, với thứ tự ưu tiên tuần tự từ trên xuống dưới.

Ví dụ:

import static org.springframework.security.authorization.AuthorityReactiveAuthorizationManager.hasRole;

// ...

@Bean

SecurityWebFilterChain springWebFilterChain(ServerHttpSecurity http) {

// @formatter:off

http

// ...

.authorizeExchange((authorize) -> authorize

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

.pathMatchers("/admin/\*\*").hasRole("ADMIN")

.pathMatchers("/db/\*\*").access((authentication, context) ->

hasRole("ADMIN").check(authentication, context)

.filter(decision -> !decision.isGranted())

.switchIfEmpty(hasRole("DBA").check(authentication, context))

)

.anyExchange().denyAll()

);

// @formatter:on

return http.build();

}

1. Có nhiều quy tắc xác quyền được chỉ đinh. Mỗi quy tắc được xem xét theo thứ tự mã chúng được khai báo.
2. Ta đã chỉ định nhiều đường dẫn url mà bất kỳ người dùng nào cũng có thể truy cập. Cụ thể, bất kỳ người dùng nào cũng có thể gửi truy cập vào url bắt đầu bằng /resources/, /signup/, /about/
3. Bất kỳ đường dẫn url nào bắt đầu bằng /admin sẽ bị hạn chế, chỉ cho phép người dùng có quyền “ROLE\_ADMIN” được phép truy cập. Bạn sẽ nhận thấy rằng vì ta đang sử dụng hasRole nên ta không cần chỉ định tiền tố “ROLE\_”.
4. Bất kỳ đường dẫn Url nào bắt đầu bằng /db/ yêu cầu người dùng phải có cả “ROLE\_ADMIN” và “ROLE\_DBA”. Điều này thể hiện tính linh hoạt của việc cung cấp một tùy chỉnh ReactiveAuthorizationManager, cho phép ta triển khai logic xác quyền tùy ý. Để đơn giản, biểu mẫu trên sử dụng lamda và để việc xác quyền cho AuthorityReactiveAuthorizationManager.hasRole. Tuy nhiên, trong tình huống thực tế, các ứng dụng có thể sẽ triển khai logic trong một lớp phù hợp như là ReactiveAuthorizationManager.
5. Bất kỳ Url không khớp với các biểu mẫu url khai báo bên trên, đều từ chối truy cập của người dùng. Đây là một chiến lược tốt, nếu ta không muốn quên cập nhật các quy tắc ủy quyền.

### EnableReactiveMethodSecurity

Spring Security hỗ trợ phương pháp bảo mật sử dụng [Reactor’s Context](https://projectreactor.io/docs/core/release/reference/), được thiết lập bằng cách sử dụng ReactiveSecurityContextHolder. Ví dụ, điều này trình bày cách truy xuất tin nhắn của người dùng hiện đang đăng nhập.

**Lưu ý**: Để điều này hoạt động, kiểu trả về của phương thức phải là một org.reactivestreams.Publisher (tức là Mono/Flux). Điều này là cần thiết để tích hợp Reactor’s Context.

Ví dụ:

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 được định nghĩa là:

Mono<String> findMessageByUsername(String username){

return Mono.just(“Hi” + username);

}

Dưới đây là phương thức cấu hình bảo mật tối thiểu của các ứng dụng reactive:

@EnableReactiveMethodSecurity

public class SecurityConfig {

@Bean

public MapReactiveUserDetailsService userDetailService () {

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);

}

}

Hãy xem xét lớp sau:

@Component

public class HelloWorldMessageService {

@PreAuthorize (“hasRole (‘ADMIN’) ”)

public Mono<String> findMessage(){

return Mono.just(“Hello World!”);

}

}

Kết hợp với cấu hình ở phía trên và @PreAuthorize(“hasRole (‘ADMIN’)”) sẽ đảm bảo rằng findByMessage chỉ được gọi bởi người dùng có vai trò ADMIN.

Điều quan trọng cần lưu ý là bất kỳ biểu thức nào trong phương thức bảo mật chuẩn đều hoạt động cho @EnableReactiveMethodSecurity. Tuy nhiên, tại thời điểm này, chúng tôi chỉ hỗ trợ kiểu trả về của Boolean hoặc boolean của biểu thức. Điều này có nghĩa là biểu thức không được chặn.

Khi tích hợp với WebFlux Security, Reactor Context được tự động thiết lập bởi Spring Security nhằm xác thực tài khoản người dùng:

@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(exchanges -> exchanges

.anyExchange().permitAll()

)

.httpBasic(withDefaults())

.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);

}

}

## OAuth2

Spring Security cung cấp tích hợp OAuth2 và WebFlux cho các ứng dụng reactive:

* Đăng nhập OAuth2: xác thực với nhà cung cấp OAuth2 hoặc OpenId Connect 1.0
* Client OAuth2: đưa ra yêu cầu tới server tài nguyên OAuth2
* OAuth2 Resource Server: bảo vệ REST endpoint sử dụng OAuth2

### Đăng nhập OAuth 2.0

Tính năng đăng nhập OAuth 2.0 cung cấp cho người dùng khả năng đăng nhập vào ứng dụng bằng tài khoản OAuth 2.0 (ví dụ: Github, …) hoặc OpenId Connect 1.0 (Ví dụ: Google)

#### Cấu hình cốt lõi

Spring Boot 2.x mang đến khả năng cấu hình tự động đầy đủ cho đăng nhập OAuth 2.0

Phần này trình bày cách cấu hình **đăng nhập WebFlux bằng OAuth 2.0** bằng Google.

**Lưu ý**: Việc triển khai OAuth 2.0 của Google để xác thực tuân theo đặc điểm kỹ thuật của OpenID Connect 1.0 và được chúng nhận OpenId.

* Làm theo phần hướng dẫn trên trang [OpenID Connect](https://developers.google.com/identity/protocols/oauth2/openid-connect), để lấy ID ứng dụng và mã bí mật.

#### Đặt URI chuyển hướng

URI chuyển hướng là đường dẫn trong ứng dụng mà người dùng cuối được chuyển hướng trở lại sau khi đã xác thực với Google và đã được cấp quyền truy cập.

Trong phần phụ “Đặt URI chuyển hướng”, hãy đảm bảo rằng trường URI chuyển hướng đucợ ủy quyền đặt thành:

* localhost:8080/login/oauth2/code/google

**Lưu ý**: Mẫu uri chuyển hướng mặc định là {baseUrl}/login/oauth2/code/{registration}. Với registerId là một định danh duy nhất cho ClientRegistration. Trong ví dụ trên, registrationId là google

**Lưu ý**: Nếu máy khách OAuth đang chạy phía sau máy chủ proxy, ta nên kiểm tra [Cấu hình server proxy](https://docs.spring.io/spring-security/reference/features/exploits/http.html), để đảm bảo ứng dụng được định cấu hình chính xác.

#### Định cấu hình

Bây giờ ta đã có ứng dụng OAuth client mới với Google, ta cần định cấu hình ứng dung để sử dụng Ứng dụng OAuth client cho quy trình xác thực.

Cấu hình file application.yml:

spring:

security:

oauth2:

client:

registration:

google:

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 máy khách OAuth
* Theo sau tiến tố thuộc tính cơ sở là Id cho ClientRegistration, chẳng hạn như google.

#### Khởi động ứng dụng

Bây giờ khi ta khởi động ứng Spring Boot 2.0, với port = 8080. Lúc này khi truy cập localhost:8080, ta sẽ thấy 1 trang đăng nhập đươc tạo mặc định, với liên kết đăng nhập bằng Google.

* Nhấp vào liên kết Google, và sau đó ta được chuyển hướng đến Google để xác thực.
* Sau khi xác thực bằng tài khoản Google, trang tiếp tục hiển thị cho ta màn hình Đồng ý. Màn hình Đồng ý yêu cầu người dùng cho phép hoặc từ chối quyền truy cập vào ứng dụng khách OAuth mã ta đã tạo trước đó. Nếu người dùng bấm vào Cho phép, thì ứng dụng khách OAuth được phép truy cập vào địa chỉ email và thông tin hồ sơ cơ bản của người dùng.

🡺 Tại thời điểm này, Ứng dung OAuth client truy cập địa chỉ email và thông tin hổ sơ cơ bản của ta từ endpoint **UserInfo** và thiết lập một session được xác thực.

#### Danh sách thuộc tính Spring Boot 2.x

Tra cứu tại: [Core Configuration :: Spring Security](https://docs.spring.io/spring-security/reference/reactive/oauth2/login/core.html)

#### CommonOAuth2Provider

CommonOAuth2Provider xác định trước một tập hợp các thuộc tính mặc định cho một số nhà cung cấp nổi tiếng như: Google, Github, Facebook, Okta

Ví dụ:

* authorization-uri
* token-uri
* user-info-uri

không thường xuyên thay đổi đối với nhà cung cấp. Do đó, việc cung cấp các giá trị mặc định để giảm cấu hình yêu cầu là rất hợp lý.

Ví dụ:

spring:

security:

oauth2:

client:

**registration**:

**google**:

client-id: <google-client-id>

client-secret: <google-client-secret>

**Lưu ý**: Tính năng tự động gán các giá trị mặc định, biết nên gán giá trị nào phụ thuộc vào {registrationId}. Trong ví dụ trên là google ~ Google, với google không phân biệt hoa thường.

Đối với các trường hợp, ta có thể muốn chỉ định một thuộc tính {registrationId} khác, chẳng hạn `google-login`, ta vẫn có thể tận dụng tính năng gán thuộc tính mặc định bằng: thuộc tính provider

spring:

security:

oauth2:

client:

registration:

google-login:

**provider**: google

client-id: google-client-id

client-secret: google-client-secret

Giải thích:

1. {registrationId} được đặt thành google-login
2. provider: google, sẽ tận dụng tính năng gán thuộc tính mặc định, được đặt trong CommonOAuth2Provider.GOOGLE.getBuilder()

**Định cấu hình thuộc tính nhà cung cấp tùy chỉnh**

Có một số nhà cung cấp OAuth 2.0 hỗ trợ nhiều nhóm người thuê, dẫn đến giao thức/miền phụ khác nhau cho mỗi người thuê.

Đối với những trường hợp này Spring Boot 2.x cung cấp thuộc tính cơ sở sau để định cấu hình các thuộc tính của nhà cung cấp tùy chỉnh:

spring:

security:

oauth2:

client:

registration:

okta:

client-id: okta-client-id

client-secret: okta-client-secret

provider:

okta:

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>

Giải thích:

1. spring.security.oauth2.client.provider.okta cho phép cấu hình tùy chỉnh các vị trị điểm cuối giao thức.

#### Tự động cấu hình

Lớp cấu hình tự động Spring boot 2.x để hỗ trợ OAuth là ReactiveOAuth2ClientAutoConfiguration

Nó thực hiện các nhiệm vụ sau:

* Đăng ký một bean ReactiveClientRegistrationRepository bao gồm: các thuộc tính của ClientRegistration để cấu hình ứng dụng khách OAuth.
* Đăng ký một bean SecurityWebFilterChain và bật đăng nhập OAuth 2.0 thông qua serverHttpSecurity.oauth2Login

Nếu ta cần ghi đề cấu hình tự động, ta cần làm những cách sau:

1. Đăng ký Bean ReactiveClientRegistrationRespository.
2. Đăng ký SecurityWebFilterChain.
3. Ghi đè hoàn toàn cấu hình tự động.

#### Đăng ký Bean ReactiveClientRegistrationRepository

Ví dụ:

@Configuration

public class OAuth2LoginConfig {

@Bean

public ReactiveClientRegistrationRepository clientRegistrationRepository() {

return new InMemoryReactiveClientRegistrationRepository (this.googleClientRegistration());

}

private ClientRegistration googleClientRegistration () {

return ClientRegistration.withRegistrationId(“google”)

.clientId(“<google-client-id>”)

.clientSecret(“<google-client-secret>”)

.clientAuthenticationMethod (ClientAuthentictionMethod.CLIENT\_SECRET\_BASIC)

.authorizationGrantType (AuthorizationGrantType.AUTHORIZATION\_CODE)

.redirectUri (“{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 (IdTokenClainNames.SUB)

.jwkSetUri (“https://www.googleapis.com/oath2/v3/certs”)

.clientName (“Google”)

.build()

}

}

#### Đăng ký Bean SecurityWebFilterChain

Ví dụ sau đay cho thấy cách đăng ký SecurityWebFilterChain bằng @EnableWebFluxSecurity và kích hoạt đăng nhập thông qua serverHttpSecurity.oauth2Login()

Ví dụ:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

.authorizeExchange(authorize -> authorize

.anyExchange().authenticated()

)

.oauth2Login(withDefaults());

return http.build();

}

}

#### Ghi đè hoàn toàn cấu hình tự động

Kết hợp 2 Bean bên trên, ta đã ghi đè hoàn toàn cấu hình tự động.

#### Cấu hình nâng cao

Khung ủy quyền OAuth 2.0 xác định điểm cuối giao thức như sau:

1. Quá trình ủy quyền sử dụng 2 điểm cuối của mấy chủ ủy quyền (tài nguyên HTTP):

* Điểm cuối ủy quyền: Được server sử dụng để nhận xác quyền từ chủ sở hữu tài nguyên thông qua chuyển hướng tác nhân người dùng.
* Điểm cuối mã thông báo: Được server sử dụng để trao đổi một khoản cấp phép lấy mã thông báo truy cập, thường là với xác thực client.

1. Cũng như một điểm cuối khách hàng:

* Điểm cuối chuyển hướng: được sử dụng bởi máy chủ ủy quyền để trả về phản hồi chứa thông tin xác thực ủy quyền cho máy khách thông qua các tác nhân người dùng của chủ sở hữu tài nguyên.

ServerHttpSecurity.oauth2Login cung cấp một số tùy chọn để tùy chình đăng nhập OAuth 2.0. Tìm hiểu chi tiết hơn, đọc code nguồn.

Các phần sau đi vào chi tiết hơn về từng tùy chọn cấu hình có sẵn:

* Trang đăng nhập OAuth 2.0
* Điểm cuối chuyển hướng
* Điểm cuối UserInfo
* Xác minh chữ ký mã thông báo.
* Đăng xuất OpenId Connect 1.0

#### Trang đăng nhập OAuth 2.0

Theo mặc định, trang đăng nhập OAuth 2.0 được tạo tự động bởi LoginPageGeneratingWebFilter.

**Lưu ý**: Để LoginPageGeneratingWebFilter hiển thị các liên kết đăng nhập OAuth, ta cần cấu hình ReactiveClientRegistrationRepository triển khai Interable <ClientRegistration>

Đường dẫn liên kết chỗ mỗi ứng dụng OAuth được mặc định như sau:

/oath2/authorization/{registrationId}

Để ghi đè đường dẫn liên kết định, hãy định cấu hình ServerHttpSecurity:

* exceptionHandling().authenticationEntryPoint
* oauth2Login().authorizationRequestResolver

Ví dụ:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

.exceptionHandling(exceptionHandling -> exceptionHandling

.authenticationEntryPoint(new RedirectServerAuthenticationEntryPoint("/login/oauth2"))

)

.oauth2Login(oauth2 -> oauth2

.authorizationRequestResolver(this.authorizationRequestResolver())

);

return http.build();

}

private ServerOAuth2AuthorizationRequestResolver authorizationRequestResolver() {

ServerWebExchangeMatcher authorizationRequestMatcher =

new PathPatternParserServerWebExchangeMatcher(

"/login/oauth2/authorization/<registrationId>");

return new DefaultServerOAuth2AuthorizationRequestResolver(

this.clientRegistrationRepository(), authorizationRequestMatcher);

}

...

}

**Lưu ý**:

* Ta cần cung cấp một @Controller với @RequestMapping (“/login/oauth2”) có khả năng hiển thi trang đăng nhập tùy chỉnh.
* ??? Sao lại là ServerHttpSecurity.exceptionHanding nhỉ ??? Tên là ngoại lệ mà lại hiển thị trang đăng nhập.
* /login/oauth2/authorization/<registrationId> là đường dẫn liên kết đăng nhập OAuth, được cấu hình bởi 1 ClientRegistration

#### Điểm cuối chuyển hướng – Redirection Endpoint

Điểm cuối chuyển hướng được **Máy chủ ủy quyền** (google, facebook, …) sử dụng để trả lại phản hồi ủy quyền (chứa thông tin xác thực ủy quyền) cho server của ta thông qua tác nhân người dùng chủ sở hữu tài nguyên.

**Lưu ý**: Đăng nhập OAuth 2.0 tận dụng Cấp mã ủy quyền. Do đó, thông tin xác thực ủy quyền là mã ủy quyền.

Điểm cuối chuyển hướng phản hồi cấp phép mặc định là /login/oauth2/code/<registratrionId>

Ví dụ, ta có thể tùy chỉnh điểm cuối phản hồi ủy quyền như sau:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

.oauth2Login(oauth2 -> oauth2

.authenticationMatcher(new PathPatternParserServerWebExchangeMatcher("/login/oauth2/callback/{registrationId}"))

);

return http.build();

}

}

**Lưu ý**: Ta cũng cần đảm bảo các kết quả ClientRegistration.redirectUri khớp với redirection endpoint

Danh sách sau đây cho thấy một ví dụ:

return CommonOAuth2Provider.GOOGLE.getBuilder("google")

.clientId("google-client-id")

.clientSecret("google-client-secret")

.redirectUri("{baseUrl}/login/oauth2/callback/{registrationId}")

.build();

#### Điểm cuối UserInfo

Điểm cuối UserInfo bao gồm 1 số tùy chỉnh cấu hình:

* Nhận/ Tạo danh sách các quyền của người dùng.
* Dịch vụ người dùng OAuth 2.0
* OpenId Connect 1.0 UserService

#### Nhận/Tạo danh sách các quyền của người dùng

Sau khi người dùng xác thực thành công với nhà cung cấp OAuth 2.0, OAuth2User.getAuthorities () hoặc OidUser.getAuthorities có thể trả về một tập hợp GrantedAuthority, sẽ được cung cấp OAuth2AuthenticationToken khi hoàn tất xác thực.

**Lưu ý**: OAuth2AuthenticationToken.getAuthorities được sử dụng để cho phép các request, truy cập vào url có hasRole (‘USER’) hoặc hasRole (‘ADMIN’)

Có một số cách để tạo danh sách quyền của người dùng:

1. Sử dụng GrantedAuthoritiesMapper
2. Chiến lược xác quyền dựa trên ReactiveOAuth2UserService

**Sử dụng GrantedAuthoritiesMapper**

Ví dụ:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

...

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public GrantedAuthoritiesMapper userAuthoritiesMapper() {

return (authorities) -> {

Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

authorities.forEach(authority -> {

if (OidcUserAuthority.class.isInstance(authority)) {

OidcUserAuthority oidcUserAuthority = (OidcUserAuthority)authority;

OidcIdToken idToken = oidcUserAuthority.getIdToken();

OidcUserInfo userInfo = oidcUserAuthority.getUserInfo();

// Map the claims found in idToken and/or userInfo

// to one or more GrantedAuthority's and add it to mappedAuthorities

} else if (OAuth2UserAuthority.class.isInstance(authority)) {

OAuth2UserAuthority oauth2UserAuthority = (OAuth2UserAuthority)authority;

Map<String, Object> userAttributes = oauth2UserAuthority.getAttributes();

// Map the attributes found in userAttributes

// to one or more GrantedAuthority's and add it to mappedAuthorities

}

});

return mappedAuthorities;

};

}

}

#### Chiến lược xác quyền dựa trên ReactiveOAuth2UserService

Tuy nhiên, chiến lược này nâng cao hơn việc sử dụng GrantedAuthoritiesMapper, nó cũng linh hoạt hơn khi cho phép ta truy cập vào **OAuth2UserRequest** và **OAuth2User** (khi sử dụng OAuth 2.0) hoặc **OidcUserRequest** và **OidcUser** (khi sử dụng OpenId Connect 1.0)

OAuth2UserRequest và OidcUserRequest cung cấp cho ta quyền vào OAuth2AccessToken, rất hữu ích trong trường hợp người ủy quyền cần tìm nạp thông tin.

Ví dụ sau cho thấy cách triển khai và định cấu hình chiến lược dựa trên xác quyền bằng cách sử dụng dịch vụ người dung OpenId connect 1.0:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

...

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public ReactiveOAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {

final OidcReactiveOAuth2UserService delegate = new OidcReactiveOAuth2UserService();

return (userRequest) -> {

// Delegate to the default implementation for loading a user

return delegate.loadUser(userRequest)

.flatMap((oidcUser) -> {

OAuth2AccessToken accessToken = userRequest.getAccessToken();

Set<GrantedAuthority> mappedAuthorities = new HashSet<>();

// TODO

// 1) Fetch the authority information from the protected resource using accessToken

// 2) Map the authority information to one or more GrantedAuthority's and add it to mappedAuthorities

// 3) Create a copy of oidcUser but use the mappedAuthorities instead

oidcUser = new DefaultOidcUser(mappedAuthorities, oidcUser.getIdToken(), oidcUser.getUserInfo());

return Mono.just(oidcUser);

});

};

}

}

#### OAuth 2.0 UserService

DefaultReactiveOAuth2UserService là triển khai của ReactiveOAuth2UserService hỗ trợ Nhà cung cấp OAuth 2.0 tiêu chuẩn.

**Lưu ý**: ReactiveOAuth2UserService **lấy các thuộc tính của người dùng cuối** (chủ sở hữu tài nguyên) từ điểm cuối UserInfo (bằng cách sử dụng mã thông báo truy cập được cấp cho máy khách trong luồng ủy quyền) và trả về một AuthenticatedPrincipal ở dạng OAuth2User.

DefaultReactiveOAuth2UserService sử dụng WebClient khi yêu cầu các thuộc tính người dùng tại điểm cuối UserInfo.

Nếu ta cần tùy chỉnh việc xử lý trước yêu cầu UserInfo hoặc xử lý sau phản hồi UserInfo, ta sẽ cần cung cấp DefaultReactiveOAuth2UserService.setWebClient một cấu hình tùy chỉnh WebClient.

Để có thể tùy chỉnh DefaultReactiveOAuth2UserService hoặc cung cấp trển khai ReactiveOAuth2UserService của riêng mình, ta cần phải định cấu hình nó như ví dụ sau:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

...

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public ReactiveOAuth2UserService<OAuth2UserRequest, OAuth2User> oauth2UserService() {

...

}

}

#### OpenID Connect 1.0 UserService

OidcReactiveOAuth2UserService là một triển khai của ReactiveOAuth2UserService hỗ trợ nhà cung cấp OpenID Connect 1.0.

OidcReactiveOAuth2UserService tận dụng DefaultReactiveOAuth2UserService khi yêu cầu các thuộc tính người dùng tại điểm cuối UserInfo.

Nếu ta cần tùy chỉnh việc xử lý trước request UserInfo hoặc xử lý sau response UserInfo, ta sẽ cần cung cấp OidcReactiveOAuth2UserService.setOauth2UserService một cấu hình tùy chỉnh ReactiveOAuth2UserService

Ví dụ:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

...

.oauth2Login(withDefaults());

return http.build();

}

@Bean

public ReactiveOAuth2UserService<OidcUserRequest, OidcUser> oidcUserService() {

...

}

}

#### ID Token Signature Verification

OpenID Connect 1.0 Authentication **ID Token**, là token bảo mật chứa tuyên bố về việc xác thực người dùng cuối bởi máy chủ cấp phép khi máy khách sử dụng.

ID Token được viết dưới dạng [Json Web Token (JWT)](https://www.rfc-editor.org/rfc/rfc7519) và phải được ký bằng [Json Web Signature](https://www.rfc-editor.org/rfc/rfc7515) (JWS).

`ReactiveOidcIdTokenDecoderFactory` cung cấp 1 `ReactiveJwtDecoder` sử dụng một `OdicIdToken` để xác minh. Thuật toán mặc định dùng để xác minh là RS256 nhưng có thể khác, bằng cách chỉ định thuật toán trong quá trình đăng ký server. Đối với trường hợp này, một trình phân giải có thể được cấu hình để trả về thuật toán JWS mong đợi của server.

Trình phân giải thuật toán JWS có tham số là 1 ClientRegistration và trả về JwsAlgorithm. Ví dụ: SignatureAlgorithm.RS256, MacAlgorithm.HS256

Đoạn mã sau đây cho thấy cách định cấu hình `OidcIdTokenDecoderFactory`, với thật toán mã hóa `MacAlgorithm.HS256` cho tất cả `ClientRegistration`:

@Bean

public ReactiveJwtDecoderFactory<ClientRegistration> idTokenDecoderFactory() {

ReactiveOidcIdTokenDecoderFactory idTokenDecoderFactory = new ReactiveOidcIdTokenDecoderFactory();

idTokenDecoderFactory.setJwsAlgorithmResolver(clientRegistration -> MacAlgorithm.HS256);

return idTokenDecoderFactory;

}

**Lưu ý**:

* Đối với các thuật toán dựa trên MAC, chẳng hạn: HS256, HS384, HS512, … thì `client-secret` tương đương với `client-id` được sử dụng làm khóa đối xứng để xác minh chứ ký.
* Nếu nhiều hơn 1 `ClientRegistration` được định cấu hình xác thực OpenID Connect 1.0. Khi này, trình phân giải thuật toán JWS có thể đánh giá `ClientRegistration` được cung cấp để xác định thuật toán nào sẽ trả về.

#### OpenID Connect 1.0 Logout

Quản lý Session kết nối OpenID 1.0 cho phép khả năng đăng xuất **Người dùng cuối** tại **Nhà cung cấp openID connect**  bằng Client. Một trong những chiến lược có sắn là [Đăng xuất bắt đầu bằng RP](https://openid.net/specs/openid-connect-session-1_0.html#RPLogout)

Nếu nhà cung cấp OpenID hỗ trợ cả quản lý session và dicovery, thì khách hàng có thể lấy từ thuộc tính `end\_session\_endpoint` của nhà cung cấp OpenID. Điều này có thể đạt được bằng cách định cấu hình `ClientRegistration` với `issuer-uri`, như trong ví dụ sau:

spring:

security:

oauth2:

client:

registration:

okta:

client-id: okta-client-id

client-secret: okta-client-secret

...

provider:

okta:

issuer-uri: <https://dev-1234.oktapreview.com>

Và `OidcClientInitiatedServerLogoutSuccessHandler`, cài đặt đăng xuất như sau:

@EnableWebFluxSecurity

public class OAuth2LoginSecurityConfig {

@Autowired

private ReactiveClientRegistrationRepository clientRegistrationRepository;

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

.authorizeExchange(authorize -> authorize

.anyExchange().authenticated()

)

.oauth2Login(withDefaults())

.logout(logout -> logout

.logoutSuccessHandler(oidcLogoutSuccessHandler())

);

return http.build();

}

private ServerLogoutSuccessHandler oidcLogoutSuccessHandler() {

OidcClientInitiatedServerLogoutSuccessHandler oidcLogoutSuccessHandler =

new OidcClientInitiatedServerLogoutSuccessHandler(this.clientRegistrationRepository);

// Sets the location that the End-User's User Agent will be redirected to

// after the logout has been performed at the Provider

oidcLogoutSuccessHandler.setPostLogoutRedirectUri("{baseUrl}");

return oidcLogoutSuccessHandler;

}

}

### OAuth 2.0 Client

Các tính năng của OAuth 2.0 Client cung cấp hỗ trợ cho vai trò client như được định nghĩa trong [Khung ủy quyền OAuth 2.0](https://www-rfc--editor-org.translate.goog/rfc/rfc6749?_x_tr_sl=auto&_x_tr_tl=vi&_x_tr_hl=vi&_x_tr_pto=wapp#section-1.1)

Ở cấp độ cao, các tính năng cốt lõi có sẵn là:

1. Hỗ trợ cấp phép.

* Mã ủy quyền
* Làm mới mã thông báo
* Thông tin xác thực của khách hàng.
* Thông tin đăng nhập mật khẩu của chủ sở hữu tài nguyên (ý chỉ người dùng cuối à).
* JWT Bearer

1. Hỗ trợ xác thực máy khách

* JWT Bearer

1. Hỗ trợ ứng dụng khách HTTP.

* `Web Client` tích hợp cho môi trường reactive (để yêu cầu tài nguyên được bảo vệ)

`ServerHttpSecurity.oauth2Client()` DSL cung cấp một số tùy chọn cấu hình để tùy chỉnh các thành phần cột lõi được OAuth 2.0 Client sử dụng.

Ví dụ:

@EnableWebFluxSecurity

public class OAuth2ClientSecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {

http

.oauth2Client(oauth2 -> oauth2

.clientRegistrationRepository(this.clientRegistrationRepository())

.authorizedClientRepository(this.authorizedClientRepository())

.authorizationRequestRepository(this.authorizationRequestRepository())

.authenticationConverter(this.authenticationConverter())

.authenticationManager(this.authenticationManager())

);

return http.build();

}

}

`ReactiveOAuth2AuthorizedClientManager` chịu trách nhiệm quản lý việc ủy quyền (hoặc ủy quyền lại) cho Ứng dụng khách OAuth 2.0, phối hợp 1 hoặc nhiều `ReactiveOAuth2AuthorizedClientProvider`

Đoạn mã sau đây là một ví dụ về cách đăng ký một `ReacticeOAuth2OauthorizedClientManager` liên kết với một tổ hợp `ReactiveOAuth2AuthorizedClientProvider`, cung cấp hỗ trợ `refresh\_token`, `client\_credentials`, `password` và các loại `authorization\_code`:

@Bean

public ReactiveOAuth2AuthorizedClientManager authorizedClientManager(

ReactiveClientRegistrationRepository clientRegistrationRepository,

ServerOAuth2AuthorizedClientRepository authorizedClientRepository) {

ReactiveOAuth2AuthorizedClientProvider authorizedClientProvider =

ReactiveOAuth2AuthorizedClientProviderBuilder.builder()

.authorizationCode()

.refreshToken()

.clientCredentials()

.password()

.build();

DefaultReactiveOAuth2AuthorizedClientManager authorizedClientManager =

new DefaultReactiveOAuth2AuthorizedClientManager(

clientRegistrationRepository, authorizedClientRepository);

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

return authorizedClientManager;

}

#### Các interface và Class cốt lõi

**ClientRegistration**

`ClientRegistration` là đại diện của khách hàng đã đăng ký với Nhà cung cấp OAuth 2.0 hoặc OpenID Connect 1.0.

Đăng ký ứng dụng khách giữ thông tin, chẳng hạn như client id, client secret, authorization grant type, redirect uri, scope, authorization uri, token uri và một vài thứ khác.

`ClientRegistration` và các thuộc tính của nó được định nghĩa như sau:

public final class ClientRegistration {

private String registrationId;

private String clientId;

private String clientSecret;

private ClientAuthenticationMethod clientAuthenticationMethod;

private AuthorizationGrantType authorizationGrantType;

private String redirectUri;

private Set<String> scopes;

private ProviderDetails providerDetails;

private String clientName;

public class ProviderDetails {

private String authorizationUri;

private String tokenUri;

private UserInfoEndpoint userInfoEndpoint;

private String jwkSetUri;

private String issuerUri;

private Map<String, Object> configurationMetadata;

public class UserInfoEndpoint {

private String uri;

private AuthenticationMethod authenticationMethod;

private String userNameAttributeName;

}

}

}

* `registrationId`: id xác định duy nhất ClientRegistration.
* `clientId`: Định danh khách hàng.
* `clientSecret`: Mã bí mật của khách hàng.
* `clientAuthenticationMethod`: phương thức được sử dụng để xác thực khách hàng với nhà cung cấp. Các giá trị được hỗ trợ là: `client\_secret\_basic`, `client\_secret\_post`, `private\_key\_jwt`, `client\_secret\_jwt và none (ý chỉ hỗ trợ cả public client)
* `authorizationGrantType`: Khung cấp phép OAuth 2.0 xác định bốn loại `Grant`. Các giá trị được hỗ trợ: `authorization\_code`, `client\_credentials`, `password` và các quyền mở rộng như là: urn:ietf:params:oauth:grant-type:jwt-bearer
* `redirectUri`: uri chuyển hướng đã đăng ký của máy khách mà Máy chủ ủy quyền, chuyển hướng tác nhân người dùng của người dùng cuối đến sau khi người cuối xác thực và cấp quyền truy cập vào máy khách.
* `scopes`: các phạm vị được khách hàng yêu cầu trong quy trình yêu cầu ủy quyền, chẳng hạn: openid, email hoặc profile
* `clientName`: Tên mô tả được sử dựng cho client. Tên có thể được sử dụng trong một số trường hợp nhất định.
* `authorizationUri`: Uri điểm cuối cấp phép cho máy chủ cấp phép.
* `tokenUri`: Uri điểm cuối cấp token cho máy chủ ủy quyền.
* `jwkSetUri`: Uri được sử dụng để truy xuất toàn bộ Json web key (JWK) từ máy chủ cấp quyền, chứa các token được sử dụng để xác minh Json Web Signature (JWS) của token ID và tùy chọn phản hồi của userInfo.
* `issuerUri`: Trả lại số nhận dạng của nhà phát hành cho nhà cung cấp OpenID Connect 1.0 hoặc máy chủ ủy quyền OAuth 2.0
* `configurationMetadata`: Thông tin cấu hình nhà cung cấp OpenID. Thông tin này chỉ có sẵn nếu thuộc tính Spring Boot 2.x: `spring.security.oauth2.client.provider.[providerId].issuerId được cấu hình.
* (userInfoEndpoint) `uri`: uri điểm cuối UserInfo được sử dụng để truy cập các xác nhận quyền sở hữu / thuộc tính của người dùng cuối đã xác thực.
* (userInfoEndPoint) `authenticationMethod`: phương thức xác thực được sử dụng khi gửi mã thông báo truy cập đến điểm cuối UserInfo. Các giá trị được hỗ trợ là `header`, `form` và `query`.
* `userNameAttributeName`: Tên của thuộc tính được trả về trong phản hồi của UserInfo, tham chiếu đến Name hoặc Identifier của người dùng cuối.

`ClientRegistration**s**` cung cấp các phương pháp thuận tiện để định cấu hình `ClientRegistration`, như trong ví dụ sau:

ClientRegistration clientRegistration =

ClientRegistrations.fromIssuerLocation("https://idp.example.com/issuer").build();

Đoạn mã trên sẽ truy vấn theo chuỗi <idp.example.com/issuer/.well-know/openid-configuration> 🡪 <idp.example.com/.well-known/openid-configuration/issuer> 🡪 <idp.example.com/.well-known/oauth-authorization-server/issuer>, dừng lại ở đoạn đầu tiên để trả về phản hồi 200.

Thay vào đó, ta có sử dụng `ClientRegistrations.fromOidcIssuerLocation()` để chỉ truy vấn điểm cuối cấu hình của nhà cung cấp kết nối OpenId

**ReactiveClientRegistrationRepository**

`ReactiveClientRegistrationRepository` phục vụ như một kho lưu trữ `ClientRegistration` cho các OAuth 2.0 / OpenID Connect 1.0

**Lưu ý**: Thông tin đăng ký của ứng dụng khách cuối cùng được lưu trữ và sở hữu bởi máy khách ủy quyền liên quan. Kho lưu trữ này cung cấp khả năng truy xuất một tập hợp con của thông tin đăng ký máy khách chính, được lưu trữ với máy chủ cấp quyền.

Spring Boot 2.x cấu hình tự động liên từng thuộc tính `spring.security.oauth2.client.registration.[registrationId] trong một thể hiện của `ClientRegistration` và sau đó biên soạn các thể hiện của `ClientRegistration ` với `ReactiveClientRegistrationRepository`.

**Lưu ý**: Lớp triển khai mặc định của `ReactiveClientRegistrationRepository` là `InMemoryReactiveClientRegistrationRepository`

**OAuth2AuthorizedClient**

`OAuth2AuthorizedClient` là đại diện của Khách hàng được ủy quyền. Máy khách được coi là được ủy quyền khi người dùng cuối (chủ sở hữu tài nguyên) đã cấp quyền cho máy khách truy cập các tài nguyên được bảo vệ.

`OAuth2AuthorizedClient` phục vụ mục đích liên kết OAuth2AccessToken (và tùy chọn OAuth2RefreshToken) với `ClientRegistration` và chủ sở hữu tài nguyên, là người dùng cuối đã cấp quyền `Principal`

**ServerOAuth2AuthorizedClientRepository** / **ReactiveOAuth2AuthorizedClientService**

`ServerOAuth2AuthorizedClientRepository` chịu trách nhiệm duy trì (các) `OAuth2AuthorizedClient` giữ các yêu cầu web.

Trong khi đó vai trò chính của `ReactiveOAuth2AuthorizedClientService` là quản lý `OAuth2AuthorizedClient` ở cấp ứng dụng

Từ góc độ nhà phát triển, `SeverOAuth2AuthorizedClientRepositoy` hoặc `ReactiveOAuth2AuthorizedClientService` cung cấp khả năng tra cứu một `OAuth2AccessToken` liên kết với một máy khách để nó có thể được sử dụng để bắt đầu yêu cầu một tài nguyên được bảo vệ.

@Controller

public class OAuth2ClientController {

@Autowired

private ReactiveOAuth2AuthorizedClientService authorizedClientService;

@GetMapping(“/”)

public Mono<String> index(Authentication authentication) {

return this.authorizedClientService.loadAuthorizedClient(“okata”, authentiaction.getName())

.map(OAuth2AuthorizedClient::getAccessToken)

…

.thenReturn(“index”)

}

}

**Lưu ý**: Spring Boot 2.x tự động cấu hình một `SeverOAuth2AuthorizedClientRepository` và `ReactiveOAuth2AuthorizedClientService`.

Triển khai mặc định của `ReactiveOAuth2AuthorizedClientService` là `InMemoryReactiveOAuth2AuthorizedClientService` lưu trữ các `OAuth2AuthorizedClient` trong bộ nhớ cứng của máy.

Ngoài ra việc triển khai R2DBC `R2dbcReactiveOAuth2AuthorizedClientService` có thể được định cấu hình để lưu các `OAuth2AuthorizedClient` trong cơ sử dữ liệu.

**Lưu ý**: `R2dbcReactiveOAuth2AuthorizedClientService` phụ thuộc vào định nghĩa bảng được mô tả trong [OAuth 2.0 Client Schema](https://docs.spring.io/spring-security/reference/servlet/appendix/database-schema.html#dbschema-oauth2-client)

**Ví dụ**:

CREATE TABLE oauth2\_authorized\_client (

client\_registration\_id varchar(100) NOT NULL,

principal\_name varchar(200) NOT NULL,

access\_token\_type varchar(100) NOT NULL,

access\_token\_value blob NOT NULL,

access\_token\_issued\_at timestamp NOT NULL,

access\_token\_expires\_at timestamp NOT NULL,

access\_token\_scopes varchar(1000) DEFAULT NULL,

refresh\_token\_value blob DEFAULT NULL,

refresh\_token\_issued\_at timestamp DEFAULT NULL,

created\_at timestamp DEFAULT CURRENT\_TIMESTAMP NOT NULL,

PRIMARY KEY (client\_registration\_id, principal\_name)

);

**ReactiveOAuth2AuthorizedClientManager / ReactiveOAuth2AuthorizedClientProvider**

`ReactiveOAuth2AuthorizedClientManger` chịu trách nhiệm quản lý **tổng thể** các `OAuth2AuthorizedClient`.

Các trách nhiệm chính bao gồm:

* Cấp phép (hoặc ủy quyền lại) ứng dụng khách OAuth 2.0, sử dụng một `ReactiveOAuth2AuthorizedClientProvider`
* Để xóa bỏ sự tồn tại của một `OAuth2AuthorizedClient`, thường sử dụng một `ReactiveOAuth2AuthorizedClientService` hoặc `ServerOAuth2AuthorizedClientRepository`
* Ủy quyền `ReactiveOAuth2AuthorizationSuccessHandler` khi một Ứng dụng khách OAuth 2.0 đã được ủy quyền thành công (hoặc ủy quyền lại).
* Ủy quyền `ReactiveOAuth2AuthorizationFailureHandler` khi ứng dụng khách OAuth 2.0 không ủy quyền (hoặc ủy quyền lại).

Triển khai một `ReactiveOAuth2AuthorizedClientProvider` cung cấp chiến lược ủy quyền (hoặc ủy quyền lại) một ứng dụng khách OAuth 2.0. Việc triển khai thường sẽ thực hiện một loại cấp phép ủy quyền, ví dụ `authorization\_code`, `client\_credentials`

Triển khai mặc định của `ReactiveOAuth2OAuthorizedClientManager` là `DefaultReactiveOAuth2AuthorizedClientManager`, được liên kết với một `ReactiveOAuth2AuthorizedClientProvider` có thể hỗ trợ nhiều loại cấp phép sử dụng , dựa trên ủy quyền. `ReactiveOAuth2AuthorizedClientProviderBuilder` có thể được sử dụng để định cấu hình và xây dựng tổ hợp dựa trên ủy quyền.

Đoạn mã sau đây, cho thấy một ví dụ về cách định cấu hình và xây dụng một tổ hợp `ReactiveOAuth2AuthorizedClientProvider` cung cấp hỗ trợ cho `authorization\_code`, `client\_credentials`, `password` và các loại `refresh\_token`:

@Bean

public ReactiveOAuth2AuthorizedClientManager authorizedClientManager(

ReactiveClientRegistrationRepository clientRegistrationRepository,

ServerOAuth2AuthorizedClientRepository authorizedClientRepository) {

ReactiveOAuth2AuthorizedClientProvider authorizedClientProvider =

ReactiveOAuth2AuthorizedClientProviderBuilder.builder()

.authorizationCode()

.refreshToken()

.clientCredentials()

.password()

.build();

DefaultReactiveOAuth2AuthorizedClientManager authorizedClientManager =

new DefaultReactiveOAuth2AuthorizedClientManager(

clientRegistrationRepository, authorizedClientRepository);

authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

return authorizedClientManager;

}

Khi một nỗ lực ủy quyền thành công, `DefautlReactiveOAuth2AuthorizedClientManager` sẽ ủy quyền `ReactiveOAuth2AuthorizationSuccessHandler`, mà theo mặc định sẽ lưu `OAuth2AuthorizedClient` thông qua `ServerOAuth2AuthorizedClientRepository`.

Trong trường hợp ủy quyền không thành công, ví dụ: `refresh\_token` không còn hợp lệ, mã `OAuth2AuthorizedClient` lưu trước đó sẽ bị xóa khỏi ServerOAuth2AuthorizedClientRepository` và `RemoveAuthorizedClientReactiveOAuth2AuthorizationFailureHandler`.

Hành vi mặc định có thể tùy chỉnh thông qua `setAuthorizationSuccessHandler(ReactiveOAuth2AuthorizationSuccessHandler)` và `setAuthorizationFailureHandler(ReactiveOAuth2AuthorizationFailureHandler)`.