Spring Security Introduction

jim stafford

Fall 2024 v2022-07-18: Built: 2024-10-29 21:01 EST

Table of Contents

Chapter 1. Introduction

Much of what we have covered to date has been focused on delivering functional capability. Before we go much further into filling in the backend parts of our application or making deployments, we need to begin factoring in security concerns. Information Security is a practice of protecting information by mitigating risks [1] Risks are identified with their impact and appropriate mitigations.

We won’t get into the details of Information Security analysis and making specific trade-offs, but we will cover how we can address the potential mitigations through the use of a framework and how that is performed within Spring Security and Spring Boot.

1.1. Goals

You will learn:

- key terms relative to implementing access control and privacy
- the flexibility and power of implementing a filter-based processing architecture
- the purpose of the core Spring Authentication components
- how to enable Spring Security
- to identify key aspects of the default Spring Security

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define identity, authentication, and authorization and how they can help protect our software system
2. identify the purpose for and differences between encoding, encryption, and cryptographic hashes
3. identify the purpose of a filter-based processing architecture
4. identify the core components within Spring Authentication
5. identity where the current user authentication is held/located
6. activate default Spring Security configuration
7. identify and demonstrate the security features of the default Spring Security configuration
8. step through a series of calls through the Security filter chain

[1] *"Information Security"*, Wikipedia

Chapter 2. Access Control

Access Control is one of the key mitigation factors within a security solution.

Identity

We need to know who the caller is and/or who is the request being made for. When you make a request in everyday life (e.g., make a pizza order) — you commonly have to supply your identity so that your request can be associated with you. There can be many layers of systems/software between the human and the action performed, so identity can be more complex than just a single value — but I will keep the examples to a simple username.

Authentication

We need verification of the requester’s identity. This is commonly something known — e.g., a password, PIN, or generated token. Additional or alternate types of authentication like something someone has (e.g., access to a specific mobile phone number or email account, or assigned token generator) are also becoming more common today and are adding a needed additional level of security to more sensitive information.

Authorization

Once we know and can confirm the identity of the requester, we then need to know what actions they are allowed to perform and information they are allowed to access or manipulate. This can be based on assigned roles (e.g., administrator, user), relative role (e.g., creator, owner, member), or releasability (e.g., access markings).

These access control decisions are largely independent of the business logic and can be delegated to the framework. That makes it much easier to develop and test business logic outside the security protections and to be able to develop and leverage mature and potentially certified access control solutions.

Chapter 3. Privacy

Privacy is a general term applied to keeping certain information or interactions secret from others. We use various encoding, encryption, and hash functions in order to achieve these goals.

3.1. Encoding

Encoding converts source information into an alternate form that is safe for communication and/or storage. [1] Two primary examples are URL and Base64 encoding of special characters or entire values. Encoding may obfuscate the data, but by itself is not encryption. Anyone knowing the encoding scheme can decode an encoded value and that is its intended purpose.

Example Base64 encoding

```
$ echo -n jim:password | base64 ①
amltOnBhc3N3b3Jk
$ echo -n amltOnBhc3N3b3Jk | base64 -d
jim:password
```

① `echo -n` echos the supplied string without new line character added - which would pollute the value

3.2. Encryption

Encryption is a technique of encoding "plaintext" information into an enciphered form ("ciphertext") with the intention that only authorized parties — in possession of the encryption/decryption keys — can convert back to plaintext. [2] Others not in possession of the keys would be forced to try to break the code through (hopefully) a significant amount of computation.

There are two primary types of keys — symmetric and asymmetric. For encryption with symmetric keys, the encryptor and decryptor must be in possession of the same/shared key. For encryption with asymmetric keys — there are two keys: public and private. Plaintext encrypted with the shared, public key can only be decrypted with the private key. SSH is an example of using asymmetric encryption.

> *Asymmetric encryption is more computationally intensive than symmetric*
>
> Asymmetric encryption is more computationally intensive than symmetric — so you may find that asymmetric encryption techniques will embed a dynamically generated symmetric key used to encrypt a majority of the payload within a smaller area of the payload that is encrypted with the asymmetric key.

Example AES Symmetric Encryption/Decryption

```
$ echo -n "jim:password" > /tmp/plaintext.txt ①
$ openssl enc -aes-256-cbc -salt -in /tmp/plaintext.txt -base64 \②
-pass pass:password > /tmp/ciphertext ③ ④
```

```
$ cat /tmp/ciphertext
U2FsdGVkX18mM2yNc337MS5r/iRJKI+roqkSym0zgMc=

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password ⑤
jim:password

$ openssl enc -d -aes-256-cbc -in /tmp/ciphertext -base64 -pass pass:password123 ⑥
bad decrypt
4611337836:error:06FFF064:digital envelope routines:CRYPTO_internal:bad decrypt
```

① the payload we will be protecting is "jim:password"

② encrypting file of plaintext with a symmetric/shared key ("password"). Result is base64 encoded.

③ the symmetric encryption key is "password"

④ "-pass" parameter supplies password and "pass:" prefix identifies password is inline in the command

⑤ decrypting file of ciphertext with valid symmetric/shared key after being base64 decoded

⑥ failing to decrypt file of ciphertext with invalid key

3.3. Cryptographic Hash

A Cryptographic Hash is a one-way algorithm that takes a payload of an arbitrary size and computes a value of a known size that is unique to the input payload. The output is deterministic such that multiple, separate invocations can determine if they were working with the same input value — even if the resulting hash is not technically the same. Cryptographic hashes are good for determining whether information has been tampered with or to avoid storing recoverable password values.

Example MD5 Cryptographic Hash without Salt

```
$ echo -n password | md5
5f4dcc3b5aa765d61d8327deb882cf99 ①
$ echo -n password | md5
5f4dcc3b5aa765d61d8327deb882cf99 ①
$ echo -n password123 | md5
482c811da5d5b4bc6d497ffa98491e38 ②
```

① Core hash algorithms produce identical results for same inputs

② Different value produced for different input

Unlike encryption there is no way to mathematically obtain the original plaintext from the resulting hash. That makes it a great alternative to storing plaintext or encrypted passwords. However, there are still some unwanted vulnerabilities by having the calculated value be the same each time.

By adding some non-private variants to each invocation (called "Salt"), the resulting values can be technically different — making it difficult to use brute force dictionary attacks. The following example uses the Apache `htpasswd` command to generate a Cryptographic Hash with a Salt value that will be different each time. The first example uses the MD5 algorithm again and the second example uses the Bcrypt algorithm — which is more secure and widely accepted for creating Cryptographic Hashes for passwords.

Example MD5 Cryptographic Hash with Salt

```
$ htpasswd -bnm jim password
jim:$apr1$ctNOftbV$SZHs/IA3ytOjx0IZEZ1w5. ①

$ htpasswd -bnm jim password
jim:$apr1$gLU9VlAl$ihDOzr8PdiCRjF3pna2EE1 ①

$ htpasswd -bnm jim password123
jim:$apr1$9sJN0ggs$xvqrmNXLq0XZWjMSN/WLG.
```

① Salt added to help defeat dictionary lookups

Example Bcrypt Cryptographic Hash with Salt

```
$ htpasswd -bnBC 10 jim password
jim:$2y$10$cBJOzUbDurA32SOSC.AnEuhUW269ACaPM7tDtD9vbrEg14i9GdGaS

$ htpasswd -bnBC 10 jim password
jim:$2y$10$RztUum5dBjKrcgiBNQlTHueqDFd60RByYgQPbugPCjv23V/RzfdVG

$ htpasswd -bnBC 10 jim password123
jim:$2y$10$s0I8X22Z1k2wK43S7dUBjup2VI1WUaJwfzX8Mg2Ng0jBxnjCEA0F2
```

htpasswd Command Line

htpasswd [options] [username] [password]

- -b: accept password from command line versus prompting for it
- -n: display results to stdout
- -B: use Bcrypt encryption
- -m: use md5 encryption
- -C: set computing time between 4 and 17; high is slower and more secure

[1] *"Code/Encoding"*,Wikipedia

[2] *"Encryption"*, Wikipedia

Chapter 4. Spring Web

Spring Framework operates on a series of core abstractions and a means to leverage them from different callchains. Most of the components are manually assembled through builders and components/beans are often integrated together through the Spring application context.

For the web specifically, the callchains are implemented through an initial web interface implemented through the hosting or embedded web server. Often the web.xml will define a certain set of filters that add functionality to the request/response flow.

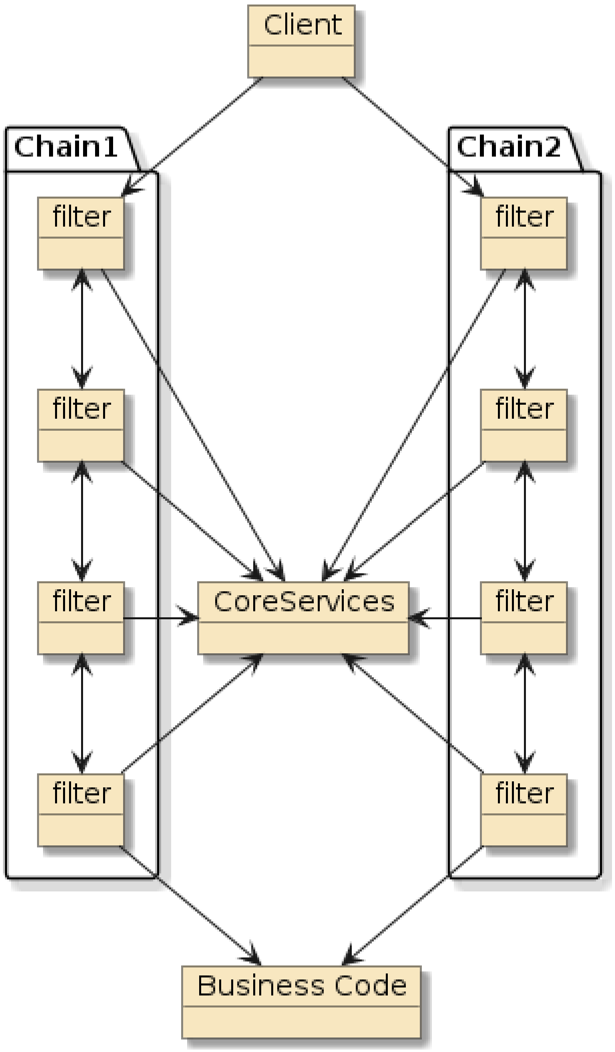


Figure 1. Spring Web Framework Operates through Flexibly Assembled Filters and Core Services

Chapter 5. No Security

We know by now that we can exercise the Spring Application Filter Chain by implementing and calling a controller class. I have implemented a simple example class that I will be using throughout this lecture. At this point in time — no security has been enabled.

5.1. Sample GET

The example controller has two example GET calls that are functionally identical at this point because we have no security enabled. The following is registered to the `/api/anonymous/hello` URI and the other to `/api/authn/hello`.

Example GET

```
@RequestMapping(path="/api/anonymous/hello",
    method= RequestMethod.GET)
public String getHello(@RequestParam(name = "name", defaultValue = "you") String name)
{
    return "hello, " + name;
}
```

We can call the endpoint using the following curl or equivalent browser call.

Calling Example GET

```
$ curl -v -X GET "http://localhost:8080/api/anonymous/hello?name=jim"
> GET /api/anonymous/hello?name=jim HTTP/1.1
< HTTP/1.1 200
< Content-Length: 10
<
hello, jim
```

5.2. Sample POST

The example controller has three example POST calls that are functionally identical at this point because we have no security or other policies enabled. The following is registered to the `/api/anonymous/hello` URI. The other two are mapped to the `/api/authn/hello` and `/api/alt/hello` URIs. [1].

Example POST

```
@RequestMapping(path="/api/anonymous/hello",
        method = RequestMethod.POST,
        consumes = MediaType.TEXT_PLAIN_VALUE,
        produces = MediaType.TEXT_PLAIN_VALUE)
public String postHello(@RequestBody String name) {
    return "hello, " + name;
}
```

We can call the endpoint using the following curl command.

Calling Example POST

```
$ curl -v -X POST "http://localhost:8080/api/anonymous/hello" \
-H "Content-Type: text/plain" -d "jim"
> POST /api/anonymous/hello HTTP/1.1
< HTTP/1.1 200
< Content-Length: 10
<
hello, jim
```

5.3. Sample Static Content

I have not mentioned it before now — but not everything served up by the application has to be live content provided through a controller. We can place static resources in the `src/main/resources/static` folder and have that packaged up and served through URIs relative to the root.

> *static resource locations*
>
> Spring Boot will serve up static content found in `/static`, `/public`, `/resources`, or `/META-INF/resources/` of the classpath.
>
> ```
> src/main/resources/
> `-- static
>     `-- content
>         `-- hello_static.txt
> ```
>
> Anything placed below `src/main/resources` will be made available in the classpath within the JAR via `target/classes`.
>
> ```
> target/classes/
> `-- static  <== classpath:/static at runtime
>     `-- content  <== /content URI at runtime
>         `-- hello_static.txt
> ```

This would be a common thing to do for css files, images, and other supporting web content. The following is a text file in our sample application.

src/main/resources/static/content/hello_static.txt

```
Hello, static file
```

The following is an example GET of that static resource file.

```
$ curl -v -X GET "http://localhost:8080/content/hello_static.txt"
> GET /content/hello_static.txt HTTP/1.1
< HTTP/1.1 200
< Content-Length: 19
<
Hello, static file
```

[1] *"Static Content"*,Spring Boot Reference Documentation

Chapter 6. Spring Security

The Spring Security framework is integrated into the web callchain using filters that form an internal Security Filter Chain.

We will look at the Security Filter Chain in more detail shortly. At this point — just know that the framework is a flexible, filter-based framework where many different authentication schemes can be enabled. Let’s take a look first at the core services used by the Security Filter Chain.

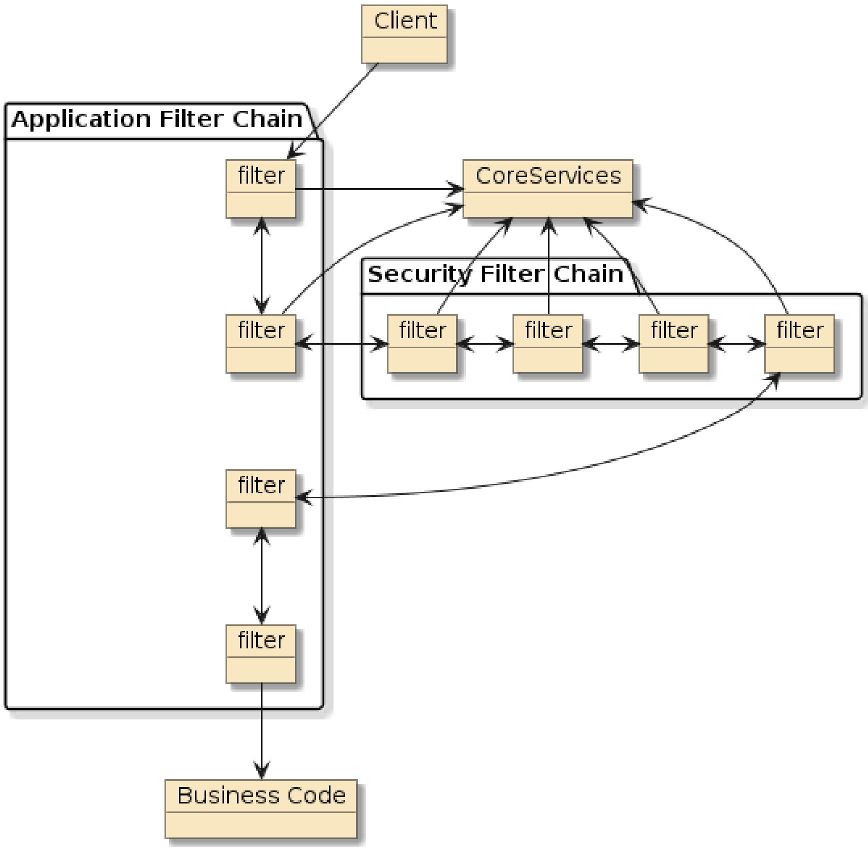


Figure 2. Spring Security Implemented as Extension of Application Filter Chain

6.1. Spring Core Authentication Framework

Once we enable Spring Security — a set of core authentication services are instantiated and made available to the Security Filter Chain. The key players are a set of interfaces with the following roles.

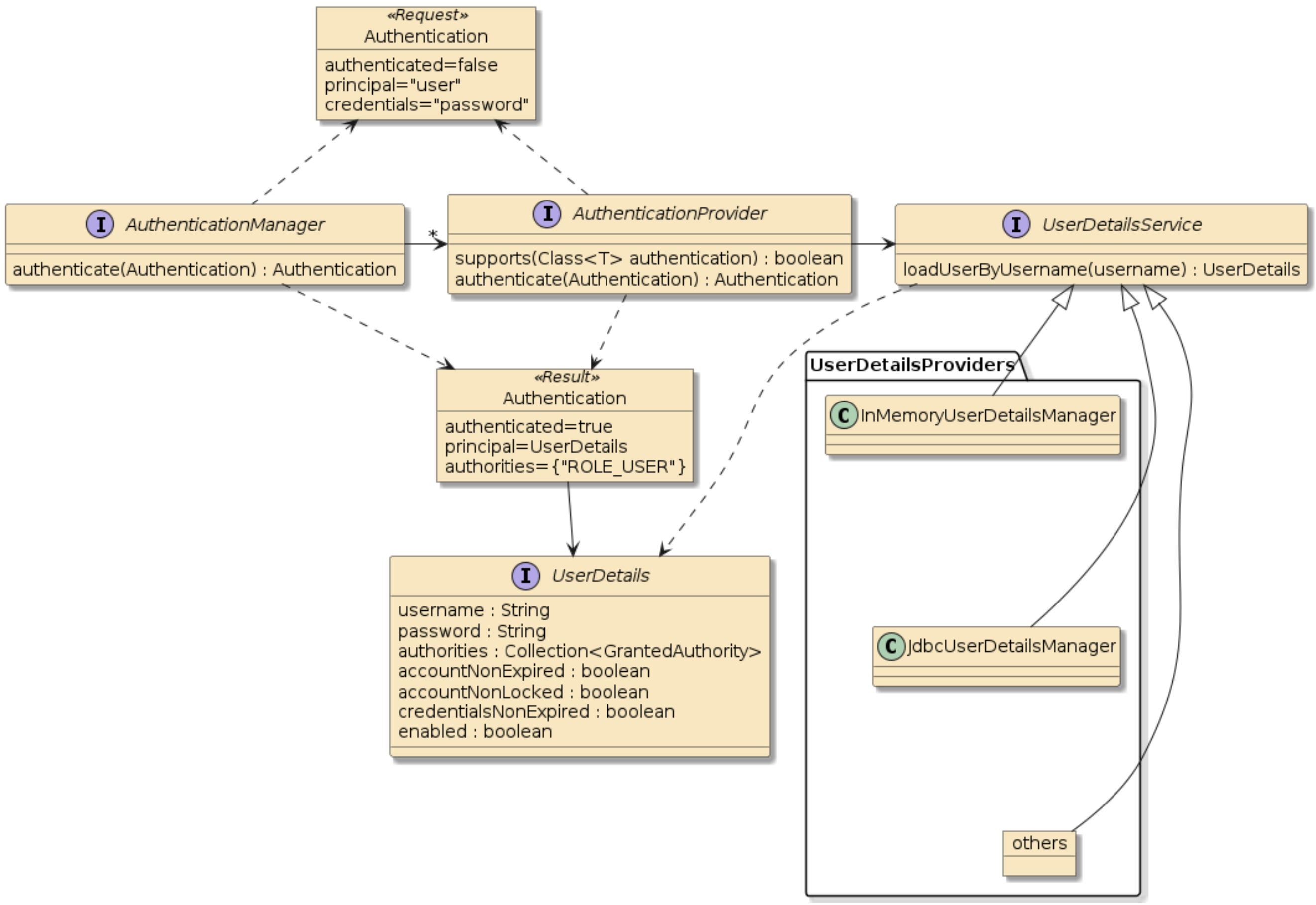

«Request»
Authentication
authenticated=false
principal="user"
credentials="password"
AuthenticationManager
authenticate(Authentication) : Authentication
*
AuthenticationProvider
supports(Class<T> authentication) : boolean
authenticate(Authentication) : Authentication
UserDetailsService
loadUserByUsername(username) : UserDetails
«Result»
Authentication
authenticated=true
principal=UserDetails
authorities={"ROLE_USER"}
UserDetails
username : String
password : String
authorities : Collection<GrantedAuthority>
accountNonExpired : boolean
accountNonLocked : boolean
credentialsNonExpired : boolean
enabled : boolean
UserDetailsProviders
InMemoryUserDetailsManager
JdbcUserDetailsManager
others

Figure 3. Spring Security Core Authentication Framework

`Authentication`

provides both an authentication request and result abstraction. All the key properties (principal, credentials, details) are defined as `java.lang.Object` to allow just about any identity and authentication abstraction be represented. For example, an `Authentication` request has a principal set to the username String and an `Authentication` response has the principal set to `UserDetails` containing the username and other account information.

`AuthenticationManager`

provides a front door to authentication requests that may be satisfied using one or more `AuthenticationProvider`

`AuthenticationProvider`

a specific authenticator with access to `UserDetails` to complete the authentication. `Authentication` requests are of a specific type. If this provider supports the type and can verify the identity claim of the caller — an `Authentication` result with additional user details is returned.

`UserDetailsService`

a lookup strategy used by `AuthenticationProvider` to obtain `UserDetails` by username. There are a few configurable implementations provided by Spring (e.g., JDBC), but we are encouraged to create our own implementations if we have a credentials repository that was not addressed.

`UserDetails`

an interface that represents the minimal needed information for a user. This will be made part of the Authentication response in the principal property.

6.2. SecurityContext

The authentication is maintained inside a SecurityContext that can be manipulated over time. The current state of authentication is located through static methods of the SecurityContextHolder class. Although there are multiple strategies for maintaining the current SecurityContext with Authentication — the most common is ThreadLocal.

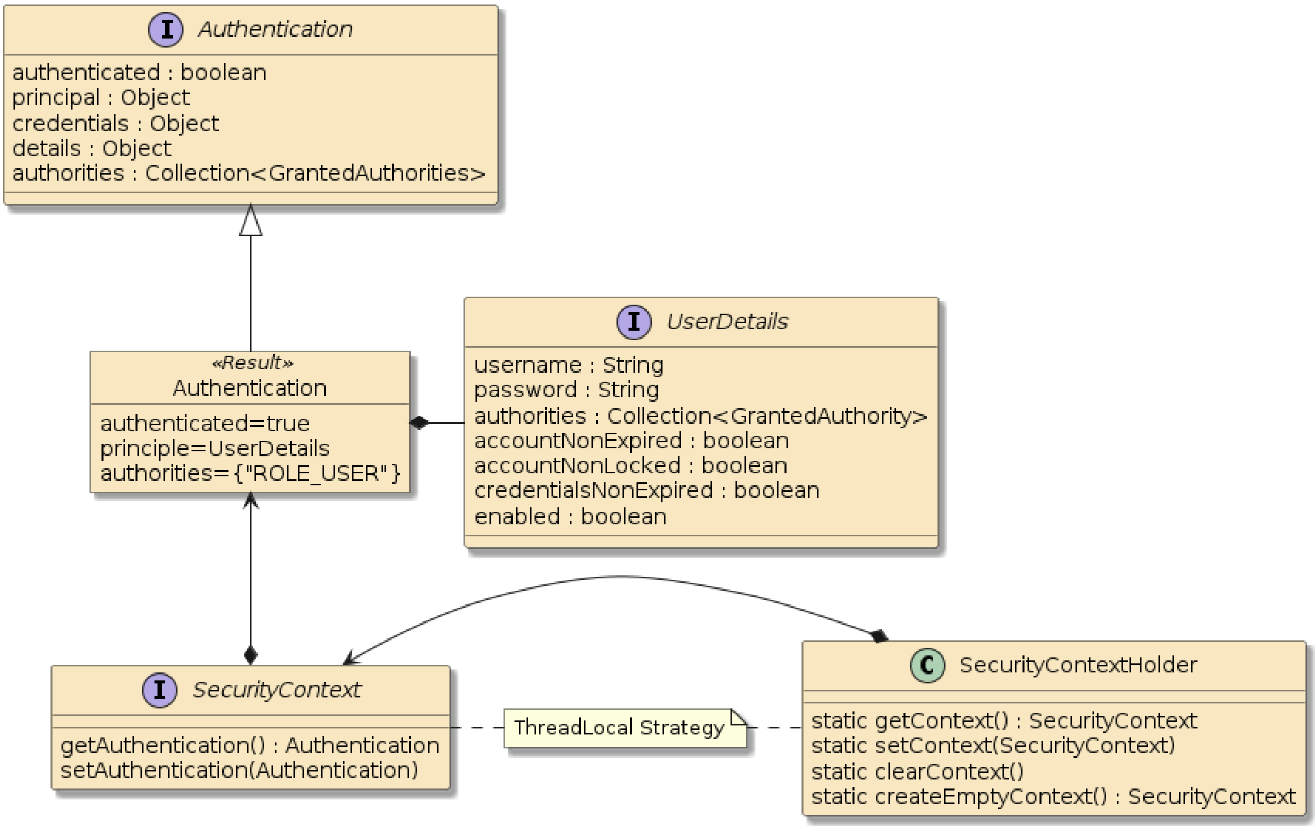

I Authentication
authenticated : boolean
principal : Object
credentials : Object
details : Object
authorities : Collection<GrantedAuthorities>
«Result»
Authentication
authenticated=true
principle=UserDetails
authorities={"ROLE_USER"}
I UserDetails
username : String
password : String
authorities : Collection<GrantedAuthority>
accountNonExpired : boolean
accountNonLocked : boolean
credentialsNonExpired : boolean
enabled : boolean
I SecurityContext
getAuthentication() : Authentication
setAuthentication(Authentication)
ThreadLocal Strategy
C SecurityContextHolder
static getContext() : SecurityContext
static setContext(SecurityContext)
static clearContext()
static createEmptyContext() : SecurityContext

Figure 4. Current SecurityContext with Authentication accessible through SecurityContextHolder

Chapter 7. Spring Boot Security AutoConfiguration

As with most Spring Boot libraries — we have to do very little to get started. Most of what you were shown above is instantiated with a single additional dependency on the `spring-boot-starter-security` artifact.

7.1. Maven Dependency

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This artifact triggers three (3) AutoConfiguration classes in the spring-boot-autoconfiguration artifact.

- For Spring Boot < 2.7, the autoconfiguration classes will be named in META-INF/spring.factories:

Spring Boot Starter Security (⇐ 2.7)

```
# org.springframework-boot:spring-boot-autoconfigure/META-INF/spring.factories
...
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
...
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
```

- For Spring Boot >= 2.7, the autoconfiguration classes will be named in META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports

Spring Boot Starter Security (>= 2.7)

```
# org.springframework-boot:spring-boot-autoconfigure/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
...
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration
```

The details of this may not be that important except to understand how the default behavior was assembled and how future customizations override this behavior.

7.2. SecurityAutoConfiguration

The `SecurityAutoConfiguration` imports two `@Configuration` classes that conditionally wire up the security framework discussed with default implementations.

- `SpringBootWebSecurityConfiguration` makes sure there is at least a default `SecurityFilterChain` (more on that later) which
    - requires all URIs be authenticated
    - activates FORM and BASIC authentication
    - enables CSRF and other security protections

    Default Security Chain Configuration

    ```
    import org.springframework.security.config.annotation.web.builders.HttpSecurity;
    import org.springframework.security.web.SecurityFilterChain;
    import static org.springframework.security.config.Customizer.withDefaults;

    @Bean
    @Order(SecurityProperties.BASIC_AUTH_ORDER) //very low priority
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
    Exception {
        http.authorizeHttpRequests((requests) -> requests.anyRequest().
    authenticated());
        http.formLogin(withDefaults());
        http.httpBasic(withDefaults());
        return http.build();
    }
    ```

- `WebSecurityEnablerConfiguration` activates all security components by supplying the `@EnableWebSecurity` annotation when the security classes are present in the classpath.

  Default Security Enabler

  ```
  @Configuration(proxyBeanMethods = false)
  @ConditionalOnMissingBean(name = BeanIds.SPRING_SECURITY_FILTER_CHAIN) ②
  @ConditionalOnClass(EnableWebSecurity.class) ①
  @EnableWebSecurity
  static class WebSecurityEnablerConfiguration {
  }
  ```

  ① activates when Spring Security is present

  ② deactivates when `springSecurityFilterChain` bean is defined

7.3. WebSecurityConfiguration

`WebSecurityConfiguration` is imported by `@EnableWebSecurity`. It gathers all the `SecurityFilterChain` beans for follow-on initialization into runtime `FilterChains`.

7.4. UserDetailsServiceAutoConfiguration

The `UserDetailsServiceAutoConfiguration` simply defines an in-memory `UserDetailsService` if one is not yet present. This is one of the provided implementations mentioned earlier — but still just a demonstration toy. The `UserDetailsService` is populated with one user:

- name: `user`, unless defined
- password: generated, unless defined

  Example Output from Generated Password

  ```
  Using generated security password: ff40aeec-44c2-495a-bbbf-3e0751568de3
  ```

Overrides can be supplied in properties

Example Default user/password Override

```
spring.security.user.name: user
spring.security.user.password: password
```

7.5. SecurityFilterAutoConfiguration

The `SecurityFilterAutoConfiguration` establishes the `springSecurityFilterChain` filter chain, implemented as a `DelegatingFilterProxy`. The delegate of this proxy is supplied by the details of the `SecurityAutoConfiguration`.

Chapter 8. Default FilterChain

When we activated Spring security we added a level of filters to the Application Filter Chain. The first was a DelegatingFilterProxy that lazily instantiated the filter using a delegate obtained from the Spring application context. This delegate ends up being a FilterChainProxy which has a prioritized list of SecurityFilterChain (implemented using DefaultSecurityFilterChain). Each SecurityFilterChain has a requestMatcher and a set of zero or more Filters. Zero filters essentially bypasses security for a particular URI pattern.

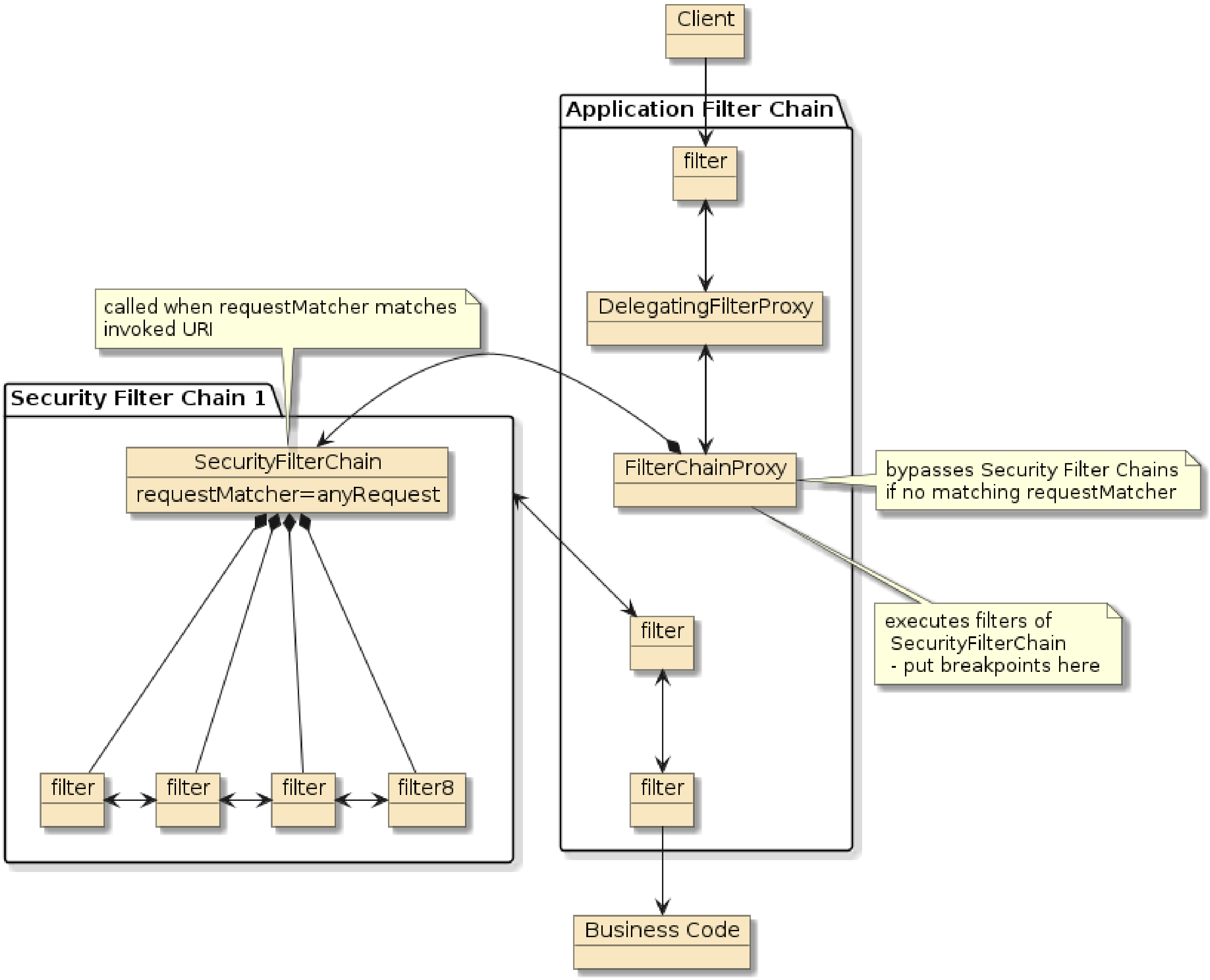

Client
Application Filter Chain
filter
DelegatingFilterProxy
FilterChainProxy
filter
filter
Business Code
called when requestMatcher matches invoked URI
Security Filter Chain 1
SecurityFilterChain
requestMatcher=anyRequest
filter
filter
filter
filter8
bypasses Security Filter Chains if no matching requestMatcher
executes filters of SecurityFilterChain - put breakpoints here

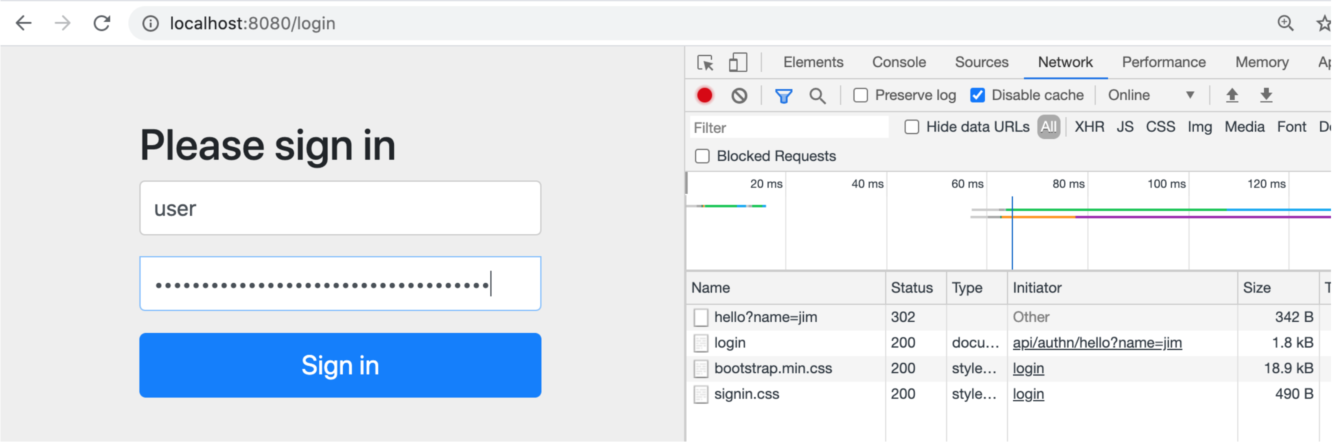
Figure 5. Default Security Filter Chain

Chapter 9. Default Secured Application

With all that said — and all we really did was add an artifact dependency to the project — the following shows where the AutoConfiguration left our application.

9.1. Form Authentication Activated

Form Authentication has been activated, and we are now stopped from accessing all URLs without first entering a valid username and password. Remember, the default username is `user` and the default password was output to the console unless we supplied one in properties. The following shows the result of a redirect when attempting to access any URL in the application.



Name	Status	Type	Initiator	Size
hello?name=jim	302		Other	342 B
login	200	docu…	api/authn/hello?name=jim	1.8 kB
bootstrap.min.css	200	style…	login	18.9 kB
signin.css	200	style…	login	490 B

Figure 6. Example Default Form Login Activated

1. We entered http://localhost:8080/api/anonymous/hello?name=jim
2. Application saw there was no authentication for the session and redirected to /login page
3. Login URL, html, and CSS supplied by spring-boot-starter-security

If we call the endpoint from curl, without indicating we can visit an HTML page, we get flatly rejected with a 401/UNAUTHORIZED. The response does inform us that BASIC Authentication is available.

Example 401/Unauthorized from Default Secured Application

```
$ curl -v http://localhost:8080/api/authn/hello?name=jim
> GET /authn/hello?name=jim HTTP/1.1
< HTTP/1.1 401
< Set-Cookie: JSESSIONID=D124368C884557286BF59F70888C0D39; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm" ①
{"timestamp":"2020-07-01T23:32:39.909+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/authn/hello"}
```

① `WWW-Authenticate` header indicates that BASIC Authentication is available

If we add an Accept header to the curl request with `text/html`, we get a 302/REDIRECT to the login page the browser automatically took us to.

Example 302/Redirect to FORM Login Page

```
$ curl -v http://localhost:8080/api/authn/hello?name=jim \
-H "Accept: text/plain,text/html" ①
> GET /authn/hello?name=jim HTTP/1.1
> Accept: text/plain, text/html
< HTTP/1.1 302
< Set-Cookie: JSESSIONID=E132523FE23FA8D18B94E3D55820DF13; Path=/; HttpOnly
< Location: http://localhost:8080/login
< Content-Length: 0
```

① adding an Accept header accepting text initiates a redirect to login form

The login (URI `/login`) and logout (URI `/logout`) forms are supplied as defaults. If we use the returned JSESSIONID when accessing and successfully completing the login form — we will continue on to our originally requested URL.

Since we are targeting APIs — we will be disabling that very soon and relying on more stateless authentication mechanisms.

9.2. Basic Authentication Activated

BASIC authentication is also activated by default. This is usable by our API out of the gate, so we will use this in examples. The following shows an example of BASIC encoding the `username:password` values in a Base64 string and then supplying the result of that encoding in an `Authorization` header prefixed with the word "BASIC ".

Example Successful Basic Authentication

```
$ echo -n user:ff40aeec-44c2-495a-bbbf-3e0751568de3 | base64
dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=

$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim \
-H "Authorization: BASIC dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM="
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=
>
< HTTP/1.1 200 ①
< Content-Length: 10
hello, jim
```

① request with successful BASIC authentication gives us the results of intended URL

Base64 web sites available if command-line tool not available

I am using a command-line tool for easy demonstration and privacy. There are various websites that will perform the encode/decode for you as well. Obviously, using a public website for real usernames and passwords would be a bad idea.

curl can Automatically Supply Authorization Header

> You can avoid the manual step of base64 encoding the username:password and manually supplying the Authorization header with curl by using the plaintext -u username:password option.
>
> ```
> curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim -u
> user:ff40aeec-44c2-495a-bbbf-3e0751568de3
> ...
> > GET /api/anonymous/hello?name=jim HTTP/1.1
> > Authorization: BASIC
> dXNlcjpmZjQwYWVlYy00NGMyLTQ5NWEtYmJiZi0zZTA3NTE1NjhkZTM=
> ...
> ```

9.3. Authentication Required Activated

If we do not supply the `Authorization` header or do not supply a valid value, we get a 401/UNAUTHORIZED status response back from the interface telling us our credentials are either invalid (did not match username:password) or were not provided.

Example Unauthorized Access

```
$ echo -n user:badpassword | base64 ②
dXNlcjpiYWRwYXNzd29yZA==

$ curl -v -X GET http://localhost:8080/api/anonymous/hello?name=jim -u
user:badpassword  ①
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpiYWRwYXNzd29yZA== ②
>
< HTTP/1.1 401
< WWW-Authenticate: Basic realm="Realm"
< Set-Cookie: JSESSIONID=32B6CDB8E899A82A1B7D55BC88CA5CBE; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm"
< Content-Length: 0
```

① bad `username:password` supplied

② demonstrating source of Authorization header

9.4. Username/Password can be Supplied

To make things more consistent during this stage of our learning, we can manually assign a username and password using properties.

src/main/resources/application.properties

```
spring.security.user.name: user
spring.security.user.password: password
```

Example Authentication with Supplied Username/Password

```
$ curl -v -X GET "http://localhost:8080/api/authn/hello?name=jim" -u user:password
> GET /api/authn/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
< HTTP/1.1 200
< Set-Cookie: JSESSIONID=7C5045AE82C58F0E6E7E76961E0AFF57; Path=/; HttpOnly
< Content-Length: 10
hello, jim
```

9.5. CSRF Protection Activated

The default Security Filter chain contains CSRF protections — which is a defense mechanism developed to prevent alternate site from providing the client browser a page that performs an unsafe (POST, PUT, or DELETE) call to an alternate site the client has as established session with. The server makes a CSRF token available to us using a GET and will be expecting that value on the next POST, PUT, or DELETE.

Example CSRF POST Rejection

```
$ curl -v -X POST "http://localhost:8080/api/authn/hello" \
-u user:password -H "Content-Type: text/plain" -d "jim"
> POST /api/authn/hello HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
> Content-Type: text/plain
> Content-Length: 3
< HTTP/1.1 401 ①
< Set-Cookie: JSESSIONID=3EEB3625749482AD9E44A3B7E25A0EE4; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm"
< Content-Length: 0
```

① POST rejected because no CSRF token provided by client

9.6. Other Headers

Spring has, by default, generated additional headers to help with client interactions that primarily have to do with common security issues.

Example Other Headers Supplied By Spring

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -u user:password
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Type: text/html;charset=UTF-8
< Content-Length: 10
< Date: Fri, 30 Aug 2024 22:16:48 GMT
<
hello, jim
```

Vary

indicates that content will be influenced by these headers. Intermediate/browser caches should especially use the Origin value when making a cache key. This prevents a different Origin from receiving a cached copy to the same resource and triggering a CORS error.

X-Content-Type-Options

informs the browser that supplied `Content-Type` header responses have been deliberately assigned [1] and to avoid Mime Sniffing (trying to determine for itself) — a problem caused by servers serving uploaded content meant to masquerade as alternate MIME types.

X-XSS-Protection

a header that informs the browser what to do in the event of a Cross-Site Scripting attack is detected. There seems to be a lot of skepticism of its value for certain browsers [2]

Cache-Control

a header that informs the client how the data may be cached. [3] This value can be set by the controller response but is set to a non-cache state by default here.

Pragma

an HTTP/1.0 header that has been replaced by Cache-Control in HTTP 1.1. [4]

Expires

a header that contains the date/time when the data should be considered stale and should be re-validated with the server. [5]

X-Frame-Options

informs the browser whether the contents of the page can be displayed in a frame. [5] This helps prevent site content from being hijacked in an unauthorized manner. This will not be pertinent to our API responses.

[1] *"X-Content-Type-Options"*, MDN web docs

[2] *"X-XSS-Protection Header"*,OWASP Cheat Sheet Series

[3] *"Cache-Control"*,MDN web docs

[4] *"Pragma"*,MDN web docs

[5] *"X-Frame-Options"*,MDN web docs

Chapter 10. Default FilterChainProxy Bean

The above behavior was put in place by the default Security AutoConfiguration — which is primarily placed within an instance of the `FilterChainProxy` class [1]. This makes the `FilterChainProxy` class a convenient place for a breakpoint when debugging security flows.

The `FilterChainProxy` is configured with a set of firewall rules that address such things as bad URI expressions that have been known to hurt web applications and zero or more SecurityFilterChains arranged in priority order (first match wins).

The default configuration has a single `SecurityFilterChain` that matches all URIs, requires authentication, and also adds the other aspects we have seen so far.

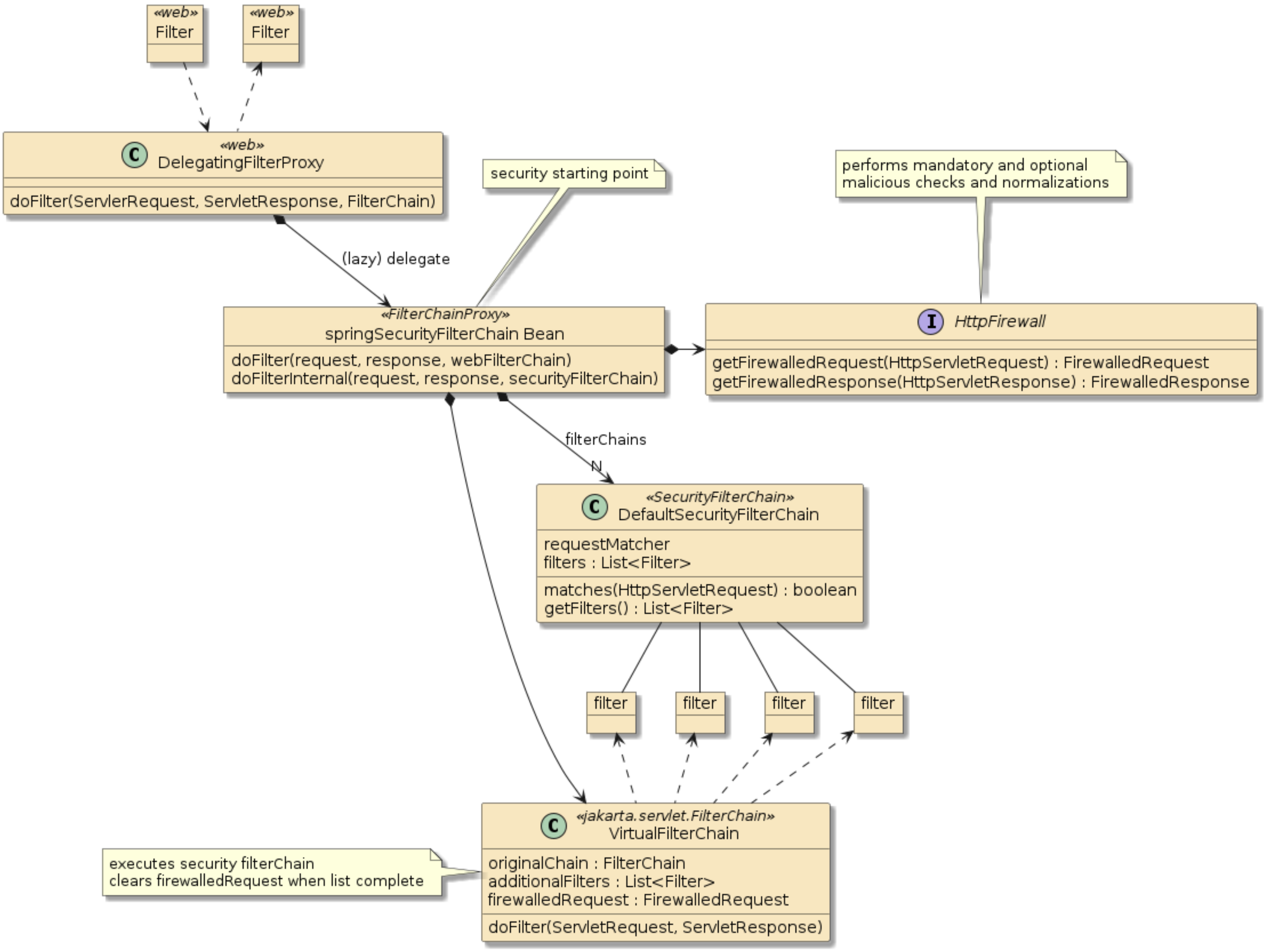

«web»
Filter
«web»
Filter
«web»
DelegatingFilterProxy
doFilter(ServlerRequest, ServletResponse, FilterChain)
(lazy) delegate
security starting point
performs mandatory and optional
malicious checks and normalizations
«FilterChainProxy»
springSecurityFilterChain Bean
doFilter(request, response, webFilterChain)
doFilterInternal(request, response, securityFilterChain)
HttpFirewall
getFirewalledRequest(HttpServletRequest) : FirewalledRequest
getFirewalledResponse(HttpServletResponse) : FirewalledResponse
filterChains
N
«SecurityFilterChain»
DefaultSecurityFilterChain
requestMatcher
filters : List<Filter>
matches(HttpServletRequest) : boolean
getFilters() : List<Filter>
filter
filter
filter
filter
«jakarta.servlet.FilterChain»
VirtualFilterChain
originalChain : FilterChain
additionalFilters : List<Filter>
firewalledRequest : FirewalledRequest
doFilter(ServletRequest, ServletResponse)
executes security filterChain
clears firewalledRequest when list complete

Figure 7. Spring FilterChainProxy Configuration

Below is a list of filters put in place by the default configuration. This — by far — is not all the available filters. I wanted to at least provide a description of the default ones before we start looking to selectively configure the chain.

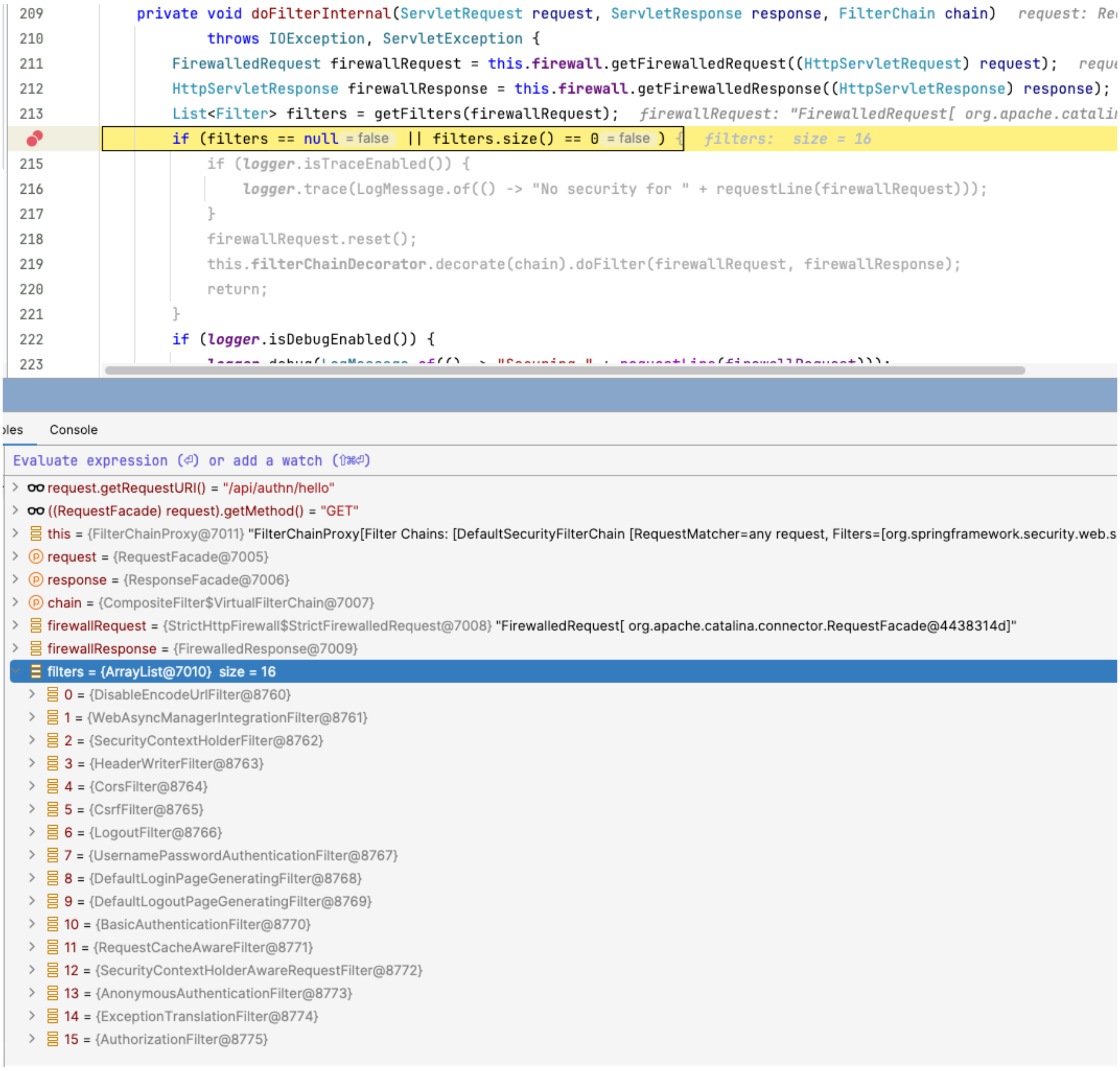
It is a pretty dry topic to just list them off. It would be best if you had the `svc/svc-security/noauthn-security-example` example loaded in an IDE with:

- the pom updated to include the `spring-boot-starter-security`

Starter Activates Default Security Policies

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

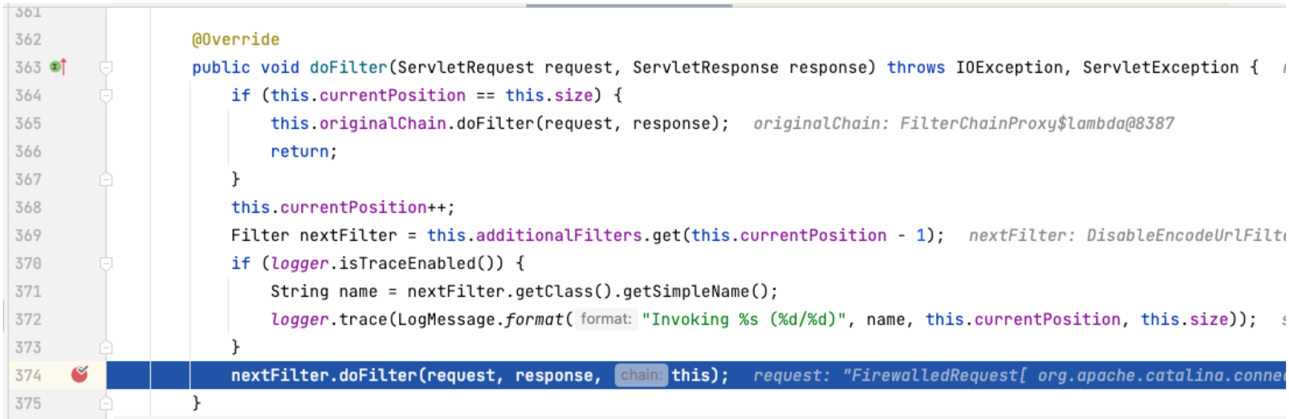
- Logger for `org.springframework.security.web` set to `TRACE`
- a breakpoint set on "FilterChainProxy.doFilterInternal()" to clearly display the list of filters that will be used for the request.



```java
    private void doFilterInternal(ServletRequest request, ServletResponse response, FilterChain chain)   request: Re
            throws IOException, ServletException {
        FirewalledRequest firewallRequest = this.firewall.getFirewalledRequest((HttpServletRequest) request);   reque
        HttpServletResponse firewallResponse = this.firewall.getFirewalledResponse((HttpServletResponse) response);
        List<Filter> filters = getFilters(firewallRequest);   firewallRequest: "FirewalledRequest[ org.apache.catali
        if (filters == null = false || filters.size() == 0 = false ) {   filters:  size = 16
            if (logger.isTraceEnabled()) {
                logger.trace(LogMessage.of(() -> "No security for " + requestLine(firewallRequest)));
            }
            firewallRequest.reset();
            this.filterChainDecorator.decorate(chain).doFilter(firewallRequest, firewallResponse);
            return;
        }
        if (logger.isDebugEnabled()) {
            logger.debug(LogMessage.of(() -> "Securing " + requestLine(firewallRequest)));
```

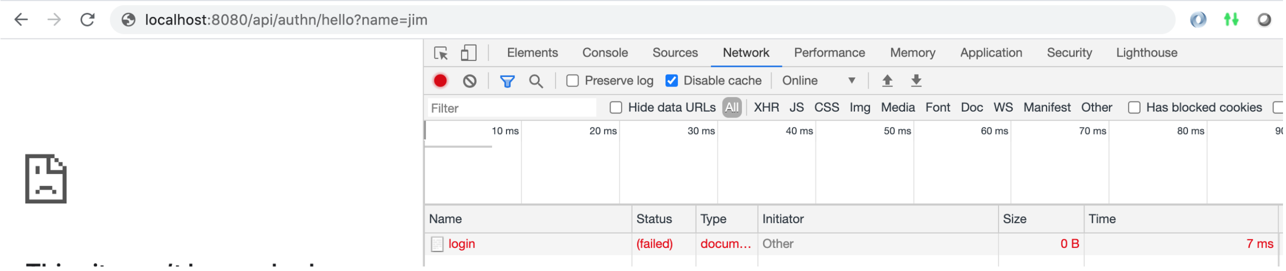
```
Evaluate expression (⏎) or add a watch (⇧⌘⏎)
> oo request.getRequestURI() = "/api/authn/hello"
> oo ((RequestFacade) request).getMethod() = "GET"
> this = {FilterChainProxy@7011} "FilterChainProxy[Filter Chains: [DefaultSecurityFilterChain [RequestMatcher=any request, Filters=[org.springframework.security.web.s
> Ⓟ request = {RequestFacade@7005}
> Ⓟ response = {ResponseFacade@7006}
> Ⓟ chain = {CompositeFilter$VirtualFilterChain@7007}
> firewallRequest = {StrictHttpFirewall$StrictFirewalledRequest@7008} "FirewalledRequest[ org.apache.catalina.connector.RequestFacade@4438314d]"
> firewallResponse = {FirewalledResponse@7009}
v filters = {ArrayList@7010}  size = 16
    > 0 = {DisableEncodeUrlFilter@8760}
    > 1 = {WebAsyncManagerIntegrationFilter@8761}
    > 2 = {SecurityContextHolderFilter@8762}
    > 3 = {HeaderWriterFilter@8763}
    > 4 = {CorsFilter@8764}
    > 5 = {CsrfFilter@8765}
    > 6 = {LogoutFilter@8766}
    > 7 = {UsernamePasswordAuthenticationFilter@8767}
    > 8 = {DefaultLoginPageGeneratingFilter@8768}
    > 9 = {DefaultLogoutPageGeneratingFilter@8769}
    > 10 = {BasicAuthenticationFilter@8770}
    > 11 = {RequestCacheAwareFilter@8771}
    > 12 = {SecurityContextHolderAwareRequestFilter@8772}
    > 13 = {AnonymousAuthenticationFilter@8773}
    > 14 = {ExceptionTranslationFilter@8774}
    > 15 = {AuthorizationFilter@8775}
```

- another breakpoint set on "FilterChainProxy.VirtualFilterChain.doFilter()" to pause in between each filter.



```
        @Override
        public void doFilter(ServletRequest request, ServletResponse response) throws IOException, ServletException {
            if (this.currentPosition == this.size) {
                this.originalChain.doFilter(request, response);   originalChain: FilterChainProxy$lambda@8387
                return;
            }
            this.currentPosition++;
            Filter nextFilter = this.additionalFilters.get(this.currentPosition - 1);   nextFilter: DisableEncodeUrlFilt
            if (logger.isTraceEnabled()) {
                String name = nextFilter.getClass().getSimpleName();
                logger.trace(LogMessage.format( format: "Invoking %s (%d/%d)", name, this.currentPosition, this.size));
            }
            nextFilter.doFilter(request, response,  chain: this);   request: "FirewalledRequest[ org.apache.catalina.conne
        }
```

- a browser open with network logging active and ready to navigate to http://localhost:8080/api/authn/hello?name=jim



localhost:8080/api/authn/hello?name=jim

Name	Status	Type	Initiator	Size	Time
login	(failed)	docum...	Other	0 B	7 ms

Whenever we make a request in the default state - we will most likely visit the following filters.

DisableEncodedUrlFilter

An exit filter that cleanses URLs provided in the response header. It specifically looks for and removes session IDs, which are not technically part of the URL and can be leaked into HTTP access logs.

WebAsyncManagerIntegrationFilter

Establishes an association between the SecurityContext (where the current caller’s credentials are held) and potential async responses making use of the `Callable` feature. Caller identity is normally unique to a thread and obtained through a `ThreadLocal`. Anything completing in an alternate thread must have a strategy to resolve the identity of this user by some other means.

SecurityContextHolderFilter

Manages SecurityContext in between calls. If appropriate — stores the SecurityContext and clears it from the call on exit. If present — restores the SecurityContext on following calls.

HeaderWriterFilter

Issues standard headers (shown earlier) that can normally be set to a fixed value and optionally overridden by controller responses.

CorsFilter

Implements Cross-Origin Resource Sharing restrictions that allow or disable exchange of information with Web clients originating from a different DNS origin (schema, domain, port). This is the source of `Vary` and `Access-Control-Allow-Origin` headers.

CsrfFilter

Checks all non-safe (POST, PUT, and DELETE) calls for a special Cross-Site Request Forgery (CSRF) token either in the payload or header that matches what is expected for the session. This attempts to make sure that anything that is modified on this site — came from this site and not a malicious source. This does nothing for all safe (GET, HEAD, OPTIONS, and TRACE) methods.

LogoutFilter

Looks for calls to log out URI. If matches, it ends the login for all types of sessions, and terminates the chain.

UsernamePasswordAuthenticationFilter

This instance of this filter is put in place to obtain the username and password submitted by the login page. Therefore, anything that is not `POST /login` is ignored. The actual `POST /login` requests have their username and password extracted, authenticated

DefaultLoginPageGeneratingFilter

Handles requests for the login URI (`POST /login`). This produces the login page, terminates the chain, and returns to caller.

DefaultLogoutPageGeneratingFilter

Handles requests for the logout URI (`GET /logout`). This produces the logout page, terminates the chain, and returns to the caller.

BasicAuthenticationFilter

Looks for BASIC Authentication header credentials, performs authentication, and continues the flow if successful or if no credentials where present. If credentials were not successful it calls an authentication entry point that handles a proper response for BASIC Authentication and ends the flow.

> *Authentication Issues?*
>
> If you are having issues with authentication, a breakpoint on line 231 of AbstractAuthenticationProcessingFilter would be a good place to start.

RequestCacheAwareFilter

This retrieves an original request that was redirected to a login page and continues it on that path.

SecurityContextHolderAwareRequestFilter

Wraps the HttpServletRequest so that the security-related calls (isAuthenticated(), authenticate(), login(), logout()) are resolved using the Spring security context.

AnonymousAuthenticationFilter

Populates the Spring security context with anonymous principal if no user is identified

ExceptionTranslationFilter

Attempts to map any thrown AccessDeniedException and AuthenticationException to an HTTP Response. It does not add any extra value if those exceptions are not thrown. This will save the current request (for access by RequestCacheAwareFilter) and commence an authentication for AccessDeniedExceptions if the current user is anonymous. The saved current request will allow the subsequent login to complete with a resumption of the original target. If FORM Authentication is active — the commencement will result in a 302/REDIRECT to the `/login` URI.

AuthorizationFilter

Restricts access based upon the response from the assigned `AuthorizationManager`.

Authorization Issues?

If you are having issues with authentication, line 95 of `AuthorizationFilter` is also a useful place to create a breakpoint.

Successfully reaching the end of the Security Filter Chain, this is where the SecurityFilter chain hands control back to the original ApplicationFilterChain, where the endpoint will get invoked.

Security FilterChain Handing Control Back to Application FilterChain

```
public void doFilter(ServletRequest request, ServletResponse response) throws
IOException, ServletException {
    if (this.currentPosition == this.size) { ①
        this.originalChain.doFilter(request, response); ②
        return;
    }
```

① successfully reached the end of the SecurityFilter chain

② delegating control back to the Application Filter chain

[1] *"FilterChainProxy"*, Spring Security Reference Manual

Chapter 11. Summary

In this module, we learned:

1. the importance of identity, authentication, and authorization within security
2. the purpose for and differences between encoding, encryption, and cryptographic hashes
3. purpose of a filter-based processing architecture
4. the identity of the core components within Spring Authentication
5. where the current user authentication is held/located
6. how to activate default Spring Security configuration
7. the security features of the default Spring Security configuration
8. to step through a series of calls through the Security filter chain for the ability to debug future access problems