Spring Security Authentication

jim stafford

Fall 2024 v2023-08-12: Built: 2024-11-19 21:34 EST

Table of Contents

Chapter 1. Introduction

In the previous example we accepted all defaults and inspected the filter chain and API responses to gain an understanding of the Spring Security framework. In this chapter we will begin customizing the authentication configuration to begin to show how and why this can be accomplished.

1.1. Goals

You will learn:

- to create a customized security authentication configurations
- to obtain the identity of the current, authenticated user for a request
- to incorporate authentication into integration tests

1.2. Objectives

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

1. create multiple, custom authentication filter chains
2. enable open access to static resources
3. enable anonymous access to certain URIs
4. enforce authenticated access to certain URIs
5. locate the current authenticated user identity
6. enable Cross-Origin Resource Sharing (CORS) exchanges with browsers
7. add an authenticated identity to RestTemplate and RestClient clients
8. add authentication to integration tests

Chapter 2. Configuring Security

To override security defaults and define a customized `FilterChainProxy`-- we must supply one or more classes that define our own `SecurityFilterChain(s)`.

2.1. WebSecurityConfigurer and Component-based Approaches

Spring has provided two ways to do this:

- `WebSecurityConfigurer`/ WebSecurityConfigurerAdapter - is the legacy, deprecated (Spring Security 5.7.0-M2; 2022), and later removed (Spring 6) definition class that acts as a modular factory for security aspects of the application. [1] There can be one-to-N `WebSecurityConfigurers` and each can define a `SecurityFilterChain` and supporting services.
- Component-based configuration - is the modern approach to defining security aspects of the application. The same types of components are defined with the component-based approach, but they are instantiated in independent `@Bean` factories. Any interdependency between the components is wired up using beans injected into the `@Bean` factory.

Early versions of Spring were based solely on the `WebSecurityConfigurer` method. Later versions of Spring 5 provided support for both. Spring 6 now only supports the Component-based method. Since you will likely encounter the `WebSecurityConfigurer` approach for a long while in older applications, I will provide some coverage of that here. However, the main focus will be the Spring 6 Component-based approach. Refer to older versions of the course examples and notes for more focused coverage of the `WebSecurityConfigurer` approach.

To highlight that the `FilterChainProxy` is populated with a prioritized list of `SecurityFilterChain` — I am going to purposely create multiple chains.

- one with the API rules (`APIConfiguration`) - highest priority
- one with the former default rules (`AltConfiguration`) - lowest priority
- one with access rules for Swagger (`SwaggerSecurity`) - medium priority

The priority indicates the order in which they will be processed and will also influence the order for the `SecurityFilterChain` s they produce. Normally I would not highlight Swagger in these examples — but it provides an additional example of how we can customize Spring Security.

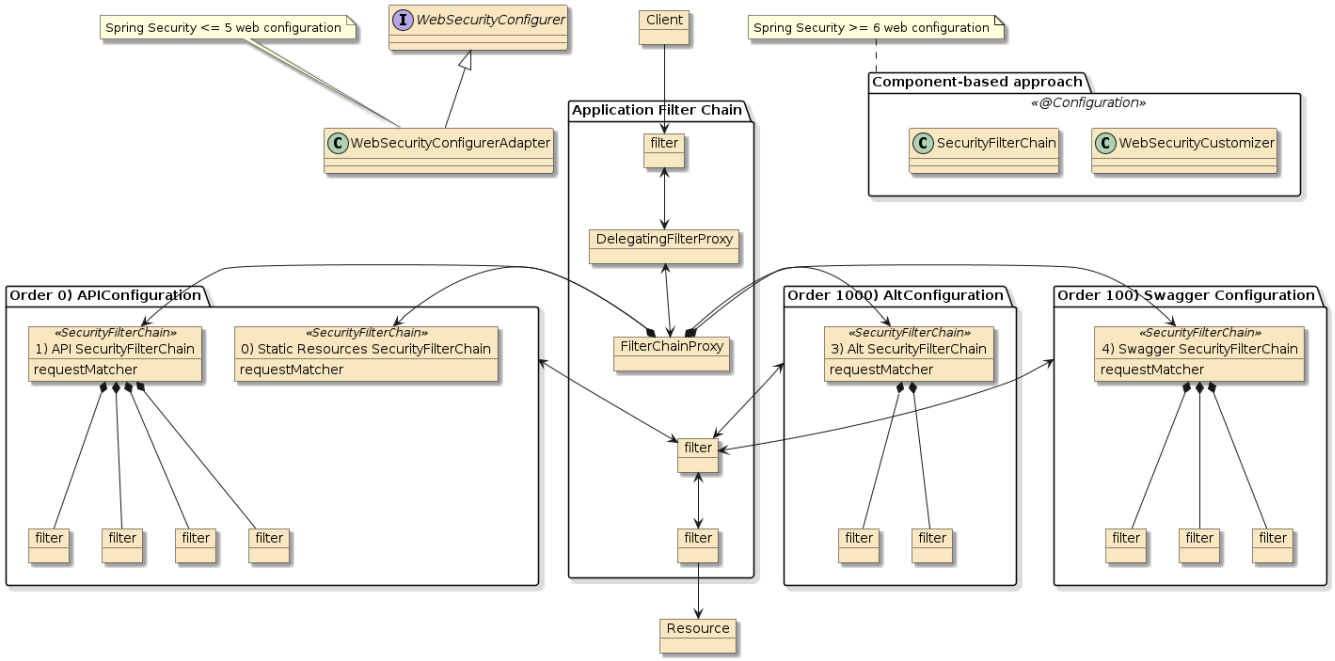


Figure 1. Multiple SecurityFilterChains

2.2. Core Application Security Configuration

The example will eventually contain several `SecurityFilterChains`, but let's start with focusing on just one of them — the "API Configuration". This initial configuration will define the configuration for access to static resources, dynamic resources, and how to authenticate our users.

2.2.1. WebSecurityConfigurerAdapter Approach

In the legacy `WebSecurityConfiguration` approach, we would start by defining a `@Configuration` class that extends `WebSecurityConfigurerAdapter` and overrides one or more of its configuration methods.

Legacy WebSecurityConfigurer no Longer Exists

Reminder - the legacy WebSecurityConfigurer approach does not exist in Spring Boot 3 / Spring Security 6. It is included here as an aid for those that may be transitioning from a legacy baseline.

WebSecurityConfigurer Approach

```
@Configuration(proxyBeanMethods = false)
@Order(0) ②
public class APIConfiguration extends WebSecurityConfigurerAdapter { ①
    @Override
    public void configure(WebSecurity web) throws Exception { ... } ③
    @Override
    protected void configure(HttpSecurity http) throws Exception { ... } ④
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception { ...
} ⑤
    @Bean
    @Override
```

```
    public AuthenticationManager authenticationManagerBean() throws Exception { ... }
⑥
```

① Create `@Configuration` class that extends `WebSecurityConfigurerAdapter` to customize `SecurityFilterChain`

② `APIConfiguration` has a high priority resulting `SecurityFilterChain` for dynamic resources

③ configure a `SecurityFilterChain` for static web resources

④ configure a `SecurityFilterChain` for dynamic web resources

⑤ optionally configure an `AuthenticationManager` for multiple authentication sources

⑥ optionally expose `AuthenticationManager` as an injectable bean for re-use in other `SecurityFilterChains`

Each `SecurityFilterChain` would have a reference to its `AuthenticationManager`. The `WebSecurityConfigurerAdapter` provided the chance to custom configure the `AuthenticationManager` using a builder.

The adapter also provided an accessor method that exposed the built `AuthenticationManager` as a pre-built component for other `SecurityFilterChains` to reference.

2.2.2. Component-based Approach

In the modern Component-based approach, we define each aspect of our security infrastructure as a separate component. These `@Bean` factory methods are within a normal `@Configuration` class that requires no inheritance. The names of the `@Bean` factory methods have no significance as long as they are unique. Only what they return has significance.

Component-based Approach

```
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCustomizer
;
import org.springframework.security.web.SecurityFilterChain;
...

@Bean
public WebSecurityCustomizer apiStaticResources() { ... } ①
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE) //0 ③
public SecurityFilterChain apiSecurityFilterChain(HttpSecurity http) throws Exception
{ ...} ②
@Bean
public AuthenticationManager authnManager(HttpSecurity http, ...) throws Exception {
⑤
    AuthenticationManagerBuilder builder = http ④
                            .getSharedObject(AuthenticationManagerBuilder.class);
    ... }
```

① define a bean to configure a `SecurityFilterChain` for static web resources

② define a bean to configure a `SecurityFilterChain` for dynamic web resources

③ high priority assigned to `SecurityFilterChain`

④ optionally configure an `AuthenticationManager` for multiple authentication sources

⑤ expose `AuthenticationManager` as an injectable bean for use in `SecurityFilterChains`

The `SecurityFilterChain` for static resources gets defined within a lambda function implementing the `WebSecurityCustomizer` interface. The `SecurityFilterChain` for dynamic resources gets directly defined by within the `@Bean` factory method.

There is no longer any direct linkage between the configuration of the `AuthenticationManager` and the `SecurityFilterChains` being built. The linkage is provided through a `getSharedObject` call of the `HttpSecurity` object that can be injected into the bean methods.

2.3. Ignoring Static Resources

One of the easiest rules to put into place is to provide open access to static content. This is normally image files, web CSS files, etc. Spring recommends **not** including dynamic content in this list. Keep it limited to static files.

Access is defined by configuring the `WebSecurity` object.

- In the legacy `WebSecurityConfigurerAdapter` approach, the modification was performed within the method overriding the `configure(WebSecurity)` method. Note that Spring 5 `WebSecurity` interface contained builder methods for `RequestMatcher` (e.g., `antMatchers()`) that no longer exist.

  Ignore Static Content Configuration - Legacy WebSecurityConfigurerAdapter approach

  ```
  import org.springframework.security.config.annotation.web.builders.WebSecurity;

  @Configuration
  @Order(0)
  public class APIConfiguration extends WebSecurityConfigurerAdapter {
      @Override
      public void configure(WebSecurity web) throws Exception {
          web.ignoring().antMatchers("/content/**");
      }
  ```

- In the modern Component-based approach, a lambda function implementing the `WebSecurityCustomizer` functional interface is returned. That lambda will be called to customize the `WebSecurity` object. Spring 6 made a breaking change to the `WebSecurity` interface by removing the approach-specific builders for matchers (e.g., `antMatchers()`) and generalized the call to `requestMatchers()`. Under the hood, by default, Spring will create a matcher that best suites the runtime environment — which will be `MvcRequestMatcher` in Spring MVC environments.

Ignore Static Content Configuration - Modern Component-based approach

```
import
org.springframework.security.config.annotation.web.configuration.WebSecurityCustomi
zer;

@Bean
public WebSecurityCustomizer apiStaticResources() {
    return (web)->web.ignoring().requestMatchers("/content/**"); ①
}
```

① delegating to Spring to create the correct matcher

WebSecurityCustomers Functional Interface

```
public interface WebSecurityCustomizer {
    void customize(WebSecurity web);
}
```

`MvcRequestMatcher` is used by Spring MVC to implement URI matching based on the actual URIs hosted within Spring MVC. The expression is still an Ant expression. It is just performed using more knowledge of the hosted resources within Spring MVC. The `MvcRequestMatcher` implementation is said to be less prone to definition error when using Spring MVC. Spring will revert to `AntRequestMatcher` when running in alternate Servlet environments (e.g., `JAX-RS`).

The generic `requestMatchers()` approach did not work when the application contained a blend of Spring MVC and non-Spring MVC (e.g., H2 Database UI) servlets. In those cases, we has to use an explicit approach documented in CVE-2023-34035. This has since been fixed.

Remember — our static content is packaged within the application by placing it under the `src/main/resources/static` directory of the source tree.

Static Content

```
$ tree src/main/resources/
src/main/resources/
|-- application.properties
`-- static
    `-- content
        |-- hello.js
        |-- hello_static.txt
        `-- index.html
$ cat src/main/resources/static/content/hello_static.txt
Hello, static file
```

With that rule in place, we can now access our static file without any credentials.

Anonymous Access to Static Content

```
$ curl -v -X GET http://localhost:8080/content/hello_static.txt
> GET /content/hello_static.txt HTTP/1.1
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Last-Modified: Fri, 03 Jul 2020 19:36:25 GMT
< Cache-Control: no-store
< Accept-Ranges: bytes
< Content-Type: text/plain
< Content-Length: 19
< Date: Fri, 03 Jul 2020 20:55:58 GMT
<
Hello, static file
```

2.4. SecurityFilterChain Matcher

The meat of the `SecurityFilterChain` definition is within the configuration of the `HttpSecurity` object.

The resulting `SecurityFilterChain` will have a `RequestMatcher` that identifies which URIs the identified rules apply to. The default is to match "any" URI. In the example below I am limiting the configuration to URIs at and below `/api/anonymous` and `/api/authn`. The matchers also allow a specific HTTP method to be declared in the definition.

- In the legacy `WebSecurityConfigurerAdapter` approach, configuration is performed in the method overriding the `configure(HttpSecurity)` method. The legacy `HttpSecurity` interface provided builders that directly supported building different types of matchers (e.g., `antMatchers()`).

  SecurityFilterChain Matcher - WebSecurityConfigurerAdapter approach

  ```
  import org.springframework.security.config.annotation.web.builders.HttpSecurity;

  @Configuration
  @Order(0)
  public class APIConfiguration extends WebSecurityConfigurerAdapter {
      @Override
      protected void configure(HttpSecurity http) throws Exception {
          http.requestMatchers(m->m.antMatchers("/api/anonymous/**","/api/authn/**")
  );①
          //... ②
      }
      ...
  ```

  ① rules within this configuration will apply to URIs below `/api/anonymous` and `/api/authn`

② `http.build()` is **not** called

This method returns `void` and the `build()` method of `HttpSecurity` should **not** be called.

- In the modern Component-based approach, the configuration is performed in a `@Bean` method that will directly return the `SecurityFilterChain`. It has the same `HttpSecurity` object injected, but note that `build()` is called within this method to return a `SecurityFilterChain`. Spring 6 made a breaking changes to:
  - the `HttpSecurity` interface by changing `requestMatchers()` to `securityMatchers()` in order to better distinguish between the pattern matching (`requestMatchers`) versus the role of the pattern matching (`securityMatchers`). The `securityMatchers()` method defines the aggregate `RequestMatchers` that are used to identify which calls invoke this `SecurityFilterChain`
  - the `AbstractRequestMatcherRegistry` interface by removing type-specific matcher builders (e.g., removed `antMatchers()`) and generalized to `requestMatchers()`

Component-based HttpSecurity Configuration

```
@Bean
@Order(0)
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.securityMatchers(m->m.requestMatchers("/api/anonymous/**","/api/authn/**"
));①
    //...
    return http.build(); ②
}
```

① rules within this configuration will apply to URIs below `/api/anonymous` and `/api/authn`

② `http.build()` is required for this `@Bean` factory

This method returns the `SecurityFilterChain` result of **calling** the `build()` method of `HttpSecurity`. This is different from the deprecated approach.

2.5. SecurityFilterChain with Explicit MvcRequestMatcher

The following lists the same solution using an explicit `MvcRequestMatcher`. The `MvcRequestMatcher` must be injected as a component because it requires knowledge of the servlet environment.

Component-based HttpSecurity Configuration with Explicit MvcRequestMatcher

```
import org.springframework.web.servlet.handler.HandlerMappingIntrospector;

@Bean
MvcRequestMatcher.Builder mvc(HandlerMappingIntrospector introspector) {
    return new MvcRequestMatcher.Builder(introspector);
}
```

```
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE) //0
public SecurityFilterChain apiSecurityFilterChain(HttpSecurity http,
        MvcRequestMatcher.Builder mvc) throws Exception {
    http.securityMatchers(m->m.requestMatchers(
            mvc.pattern("/api/anonymous/**"),
            mvc.pattern("/api/authn/**")
            ));
```

2.6. HttpSecurity Builder Methods

The `HttpSecurity` object is "builder-based" and has several options on how it can be called.

- `http.methodAcceptingBuilt(builtObject)`
- `http.methodReturningBuilder().customizeBuilder()` (non-lambda)
- `http.methodPassingBuilderToLambda(builder→builder.customizeBuilder())`

2.6.1. Deprecated non-Lambda Methods

Spring Security 6 has deprecated the non-lambda customize approach (for removal in Spring Security 7) in favor of the lambda approach. [2] The following non-lambda approach (supplying no options) will result in a deprecation warning.

Deprecated non-Lambda Customizer Approach

```
http.httpBasic();
```

Spring Security has added a `Customizer.withDefaults()` construct that can mitigate the issue when supplying no customizations.

Lambda Customization Approach

```
import org.springframework.security.config.Customizer;

http.httpBasic(Customizer.withDefaults());
```

2.6.2. Chaining not Necessary

The builders are also designed to be chained. It is quite common to see the following syntax used.

Chained Builder Calls

```
http.authorizeRequests(cfg->cfg.anyRequest().authenticated())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());
```

As much as I like chained builders — I am not a fan of that specific syntax when starting out. Especially if we are experimenting and commenting/uncommenting configuration statements. We can simply make separate calls. You will see me using separate calls. Either style functionally works the same.

Separate Builder Calls with Lambdas

```
http.authorizeRequests(cfg->cfg.anyRequest().authenticated());
http.formLogin(Customizer.withDefaults());
http.httpBasic(Customizer.withDefaults());
```

2.7. Match Requests

We first want to scope our `HttpSecurity` configuration commands using one of the securityMatcher methods. We can provide a `RequestMatcher` using `securityMatcher()` or a configurer using `securityMatchers()`. The following will create a `RequestMatcher` that will match any HTTP method and URI below `/api/anonymous` and `/api/authn`. We can add an `Http.(METHOD)` if we want it to be specific to only that HTTP method.

Using Default Request Matchers

```
http.securityMatchers(m->m.requestMatchers("/api/anonymous/**","/api/authn/**")); ①
//http.securityMatcher("/api/anonymous/**","/api/authn/**"); ②
```

① matches all HTTP methods matching the given URI patterns

② shortcut when no HttpMethod is needed

RequestMatchers can be Aggregated

The code above technically creates three (3) `RequestMatchers`. Two (2) to match the individual (method)/URIs and one (1) parent to aggregate them into an "or" predicate.

The `requestMatchers()` configurer (*in non-ambiguous environments) will create an `MvcRequestMatcher` under the hood when Spring MVC is present and an `AntRequestMatcher` for other servlet frameworks. The `MvcRequestMatcher` is said to be less prone to error (i.e., missed matches) than other techniques. Spring 6 took away builder methods to directly create the `AntRequestMatcher` (and other explicit types), but they can still be created manually. This technique can be used to create any type of RequestMatcher.

Using Explicit Request Matchers

```
import org.springframework.security.web.util.matcher.OrRequestMatcher;
import org.springframework.security.web.util.matcher.RegexRequestMatcher;

http.securityMatcher(new OrRequestMatcher(②
        RegexRequestMatcher.regexMatcher("^/api/anonymous$"),①
        RegexRequestMatcher.regexMatcher("^/api/anonymous/.*"),
        RegexRequestMatcher.regexMatcher("^/api/authn$"),
```

```
        RegexRequestMatcher.regexMatcher("^/api/authn/.*")
));
```

① supplying an explicit type of `RequestMatcher`

② Spring Security is aggregating the list into an `OrRequestMatcher`

Notice the `requestMatcher` is the primary property in the individual chains and the rest of the configuration is impacting the filters within that chain.

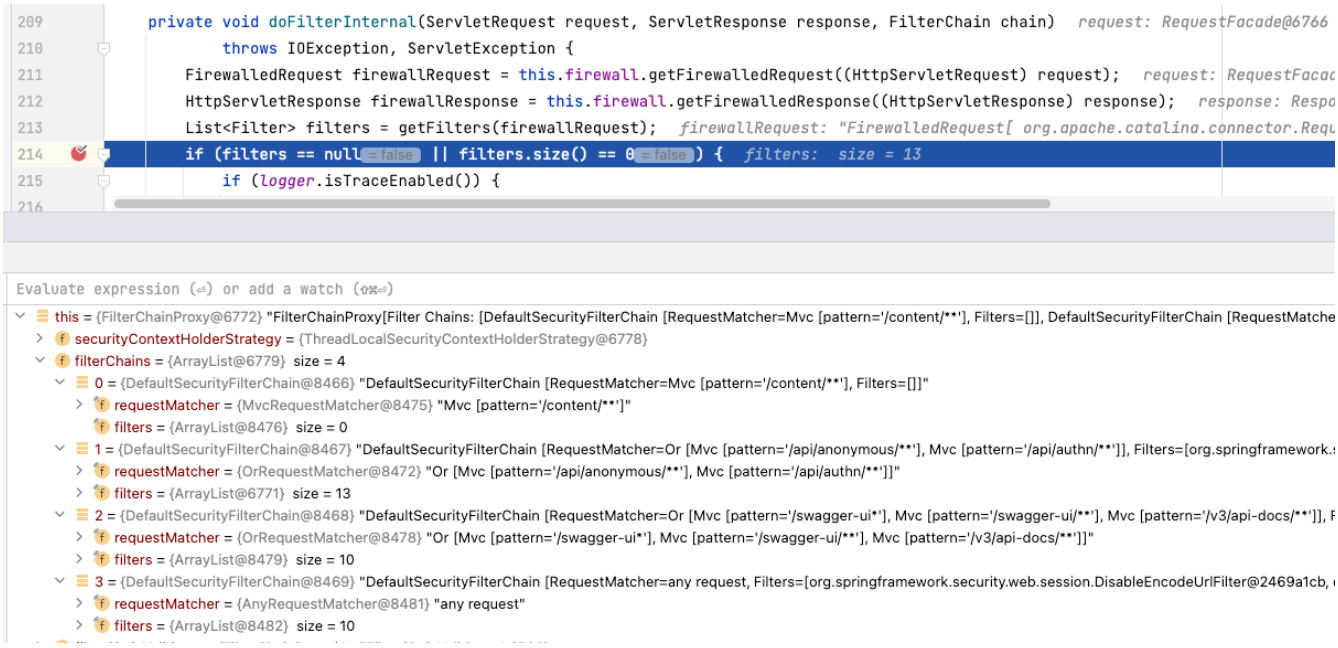


Figure 2. Request Matchers

Notice also that our initial `SecurityFilterChain` is within the other chains in the example and is high in priority because of our `@Order` value assignment:

2.8. Authorize Requests

Next I am showing the authentication requirements of the `SecurityFilterChain`. Calls to the `/api/anonymous` URIs do not require authentication. Calls to the `/api/authn` URIs do require authentication.

Defining Authentication Requirements

```
http.authorizeHttpRequests(cfg->cfg.requestMatchers("/api/anonymous/**").permitAll());
http.authorizeHttpRequests(cfg->cfg.anyRequest().authenticated());
```

The permissions off the matcher include:

- permitAll() - no constraints
- denyAll() - nothing will be allowed
- authenticated() - only authenticated callers may invoke these URIs
- role restrictions that we won't be covering just yet

You can also make your matcher criteria method-specific by adding in a `HttpMethod` specification.

Matchers Also Supports HttpMethod Criteria

```
import org.springframework.http.HttpMethod;
...
...(cfg->cfg.requestMatchers(HttpMethod.GET,"/api/anonymous/**").permitAll())
...(cfg->cfg.requestMatchers(HttpMethod.GET).permitAll())
```

requestMatchers are evaluated after satisfying securityMatcher(s)

RequestMatchers are evaluated within the context of what satisfied the filter's securityMatcher(s). If you form a Security FilterChain for a specific base URI, the requestMatcher is only defining rules for what that chain processes.

2.9. Authentication

In this part of the example, I am enabling BASIC Auth and eliminating FORM-based authentication. For demonstration only — I am providing a custom name for the realm name returned to browsers.

```
http.httpBasic(cfg->cfg.realmName("AuthConfigExample")); ①
http.formLogin(cfg->cfg.disable());
```

Realm name is not a requirement to activate Basic Authentication. It is shown here solely as an example of something easily configured.

Example Realm Header used in Authentication Required Response

```
< HTTP/1.1 401
< WWW-Authenticate: Basic realm="AuthConfigExample" ①
```

① Realm Name returned in HTTP responses requiring authentication

2.10. Header Configuration

In this portion of the example, I am turning off two of the headers that were part of the default set: XSS protection and frame options. There seemed to be some debate on the value of the XSS header [3] and we have no concern about frame restrictions. By disabling them — I am providing an example of what can be changed.

`CSRF` protections have also been disabled to make non-safe methods more sane to execute at this time. Otherwise, we would be required to supply a value in a POST that came from a previous GET (all maintained and enforced by optional filters).

Header Configuration

```
http.headers(cfg->{
```

```
    cfg.xssProtection(xss-> xss.disable());
    cfg.frameOptions(fo->fo.disable());
});
http.csrf(cfg->cfg.disable());
```

2.11. Stateless Session Configuration

I have no interest in using the Http Session to maintain identity between calls — so this should eliminate the `SET-COOKIE` commands for the `JSESSIONID`.

Stateless Session Configuration

```
http.sessionManagement(cfg->
    cfg.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

[1] *"Spring Security without the WebSecurityConfigurerAdapter"*, Spring.io, Feb 21, 2022

[2] *"Spring Security, Preparing for 7.0, Configuration Methods"*,Spring.io

[3] *"X-XSS-Protection"*,MDN Web Docs

Chapter 3. Configuration Results

With the above configurations in place — we can demonstrate the desired functionality and trace the calls through the filter chain if there is an issue.

3.1. Successful Anonymous Call

The following shows a successful anonymous call and the returned headers. Remember that we have gotten rid of several unwanted features with their headers. The controller method has been modified to return the identity of the authenticated caller. We will take a look at that later — but know the source of the additional `:caller=` string was added for this wave of examples.

Successful Anonymous Call

```
$ 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
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 25
< Date: Fri, 03 Jul 2020 22:11:11 GMT
<
hello, jim :caller=(null) ①
```

① we have no authenticated user

3.2. Successful Authenticated Call

The following shows a successful authenticated call and the returned headers.

Successful Authenticated Call

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password ①
> GET /api/authn/hello?name=jim HTTP/1.1
> Authorization: BASIC dXNlcjpwYXNzd29yZA==
< HTTP/1.1 200
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 23
< Date: Fri, 03 Jul 2020 22:12:34 GMT
<
hello, jim :caller=user ②
```

① example application configured with username/password of `user/password`

② we have an authenticated user

3.3. Rejected Unauthenticated Call Attempt

The following shows a rejection of an anonymous caller attempting to invoke a URI requiring an authenticated user.

Rejected Unauthenticated Call Attempt

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim ①
> GET /api/authn/hello?name=jim HTTP/1.1
< HTTP/1.1 401
< WWW-Authenticate: Basic realm="AuthConfigExample"
< X-Content-Type-Options: nosniff
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Fri, 03 Jul 2020 22:14:20 GMT
<
{"timestamp":"2020-07-03T22:14:20.816+00:00","status":401,
"error":"Unauthorized","message":"Unauthorized","path":"/api/authn/hello"}
```

① attempt to make anonymous call to authentication-required URI

Chapter 4. Authenticated User

Authenticating the identity of the caller is a big win. We likely will want their identity at some point during the call.

4.1. Inject UserDetails into Call

One option is to inject the `UserDetails` containing the username (and authorities) for the caller. Methods that can be called without authentication will receive the `UserDetails` if the caller provides credentials but must protect itself against a null value if actually called anonymously.

Injecting Caller Identity into Controller

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.UserDetails;
...
public String getHello(@RequestParam(name = "name", defaultValue = "you") String name,
                       @AuthenticationPrincipal UserDetails user) {
    return "hello, " + name + " :caller=" + (user==null ? "(null)" : user.getUsername
());
}
```

4.2. Obtain SecurityContext from Holder

The other option is to look up the `UserDetails` through the `SecurityContext` stored within the `SecurityContextHolder` class. This allows any caller in the call flow to obtain the identity of the caller at any time.

Obtaining Caller Identity from SecurityContextHolder

```
import org.springframework.security.core.context.SecurityContextHolder;

public String getHelloAlt(@RequestParam(name = "name", defaultValue = "you") String
name) {
    Authentication authentication = SecurityContextHolder.getContext()
.getAuthentication();
    Object principal = null!=authentication ? authentication.getPrincipal() : "(null)
";
    String username = principal instanceof UserDetails ?
        ((UserDetails)principal).getUsername() : principal.toString();
    return "hello, " + name + " :caller=" + username;
}
```

Chapter 5. Swagger BASIC Auth Configuration

Once we enabled default security on our application — we lost the ability to fully utilize the Swagger page. We did not have to create a separate `SecurityFilterChain` for just the Swagger endpoints — but doing so provides some nice modularity and excuse to further demonstrate Spring Security configurability.

Check Defaults

Spring Boot used to always apply a default filter with `authenticated()`, denying access to any URI lacking a matching `securityMatcher`. Spring Boot 3.3.2 no longer applies a default filter when one is provided (as we did above) — leaving it unevaluated, resulting in open access and other defaults. Swagger UI can be accessed in this setting, but the BASIC authentication within the page is inoperable. Verify default behavior in addition to the URIs of importance to your application.

I have added a separate security configuration for the OpenAPI and Swagger endpoints.

5.1. Swagger Authentication Configuration

The following configuration allows the OpenAPI and Swagger endpoints to be accessed anonymously and handle authentication within OpenAPI/Swagger.

- Swagger SecurityFilterChain using the legacy `WebSecurityConfigurerAdapter` approach

```
@Configuration(proxyBeanMethods = false)
@Order(100) ①
public class SwaggerSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers(cfg->cfg
                .antMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
        http.authorizeRequests(cfg->cfg.anyRequest().permitAll());
        http.csrf().disable();
    }
}
```

① Priority (100) is after core application (0) and prior to default rules (1000)

- Swagger SecurityFilterChain using the modern Component-based approach

```
@Bean
@Order(100) ①
public SecurityFilterChain swaggerSecurityFilterChain(HttpSecurity http) throws
Exception {
    http.securityMatchers(cfg->cfg
```

```
            .requestMatchers("/swagger-ui*", "/swagger-ui/**", "/v3/api-docs/**"));
    http.authorizeHttpRequests(cfg->cfg.anyRequest().permitAll());
    http.csrf(cfg->cfg.disable());
    return http.build();
}
```

① Priority (100) is after core application (0) and prior to default rules (1000)

5.2. Swagger Security Scheme

In order for Swagger to supply a username:password using BASIC Auth, we need to define a `SecurityScheme` for Swagger to use. The following bean defines the core object the methods will be referencing.

Swagger BASIC Auth Security Scheme

```
package info.ejava.examples.svc.authn;

import io.swagger.v3.oas.models.Components;
import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.security.SecurityScheme;
import org.springframework.context.annotation.Bean;
...

@Bean
public OpenAPI customOpenAPI() {
    return new OpenAPI()
            .components(new Components()
                    .addSecuritySchemes("basicAuth",
                            new SecurityScheme()
                                    .type(SecurityScheme.Type.HTTP)
                                    .scheme("basic")));
}
```

The `@Operation` annotations can now reference the `SecuritySchema` to inform the SwaggerUI that BASIC Auth can be used against that specific operation. Notice too, that we needed to make the injected `UserDetails` optional — or even better — hidden from OpenAPI/Swagger since it is not part of the HTTP request.

Swagger Operation BASIC Auth Definition

```
package info.ejava.examples.svc.authn.authcfg.controllers;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.Parameter;

@RestController
public class HelloController {
...
```

```
    @Operation(description = "sample authenticated GET",
                  security = @SecurityRequirement(name="basicAuth")) ①
    @RequestMapping(path="/api/authn/hello",
            method= RequestMethod.GET)
    public String getHello(
            @RequestParam(name="name",defaultValue="you",required=false) String name,
            @Parameter(hidden = true) ②
            @AuthenticationPrincipal UserDetails user) {
        return "hello, " + name + " :caller=" + user.getUsername();
    }
```

① added `@SecurityRequirement` to operation to express within OpenAPI that this call accepts Basic Auth

② Identified parameter as not applicable to HTTP callers

With the `@SecurityRequirement` in place, the Swagger UI provides a means to supply username/password for subsequent calls.

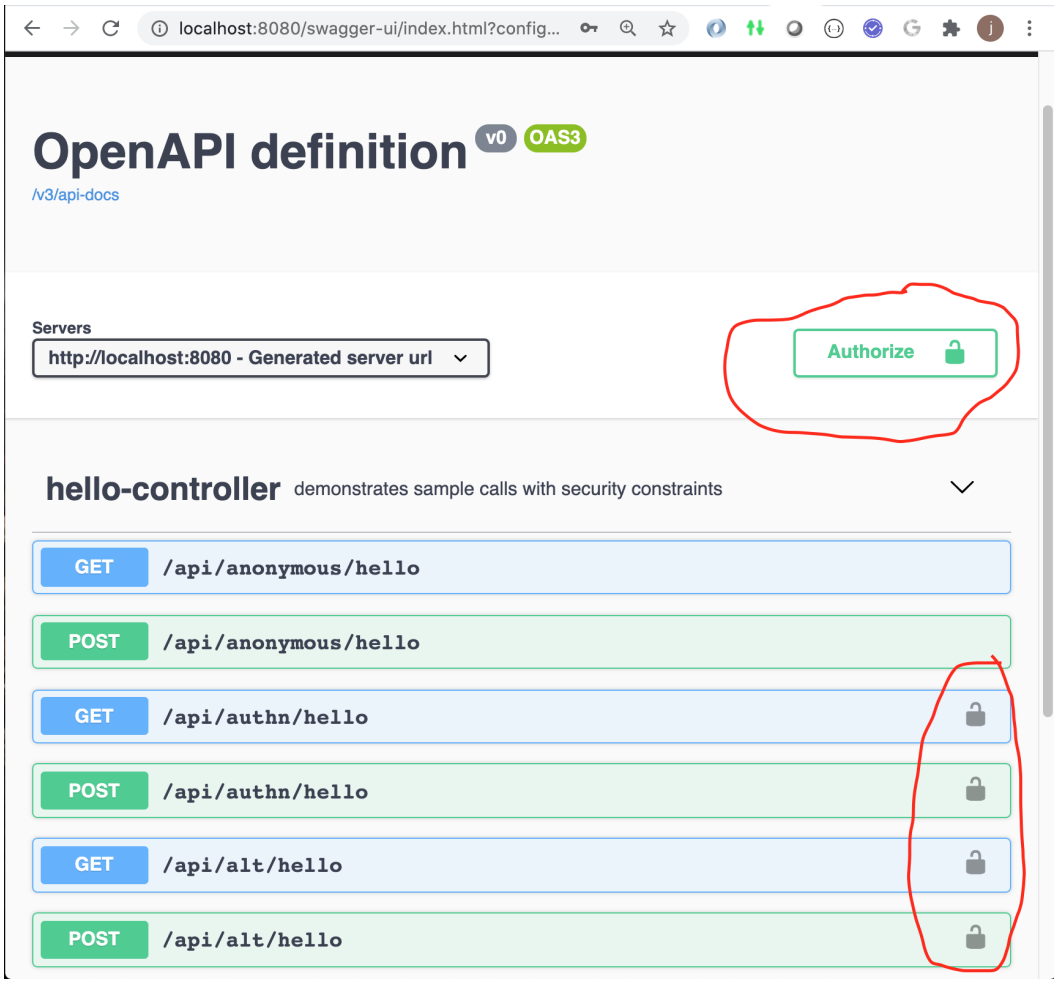


Figure 3. Swagger with BASIC Auth Configured

When making a call — Swagger UI adds the Authorization header with the previously entered credentials.

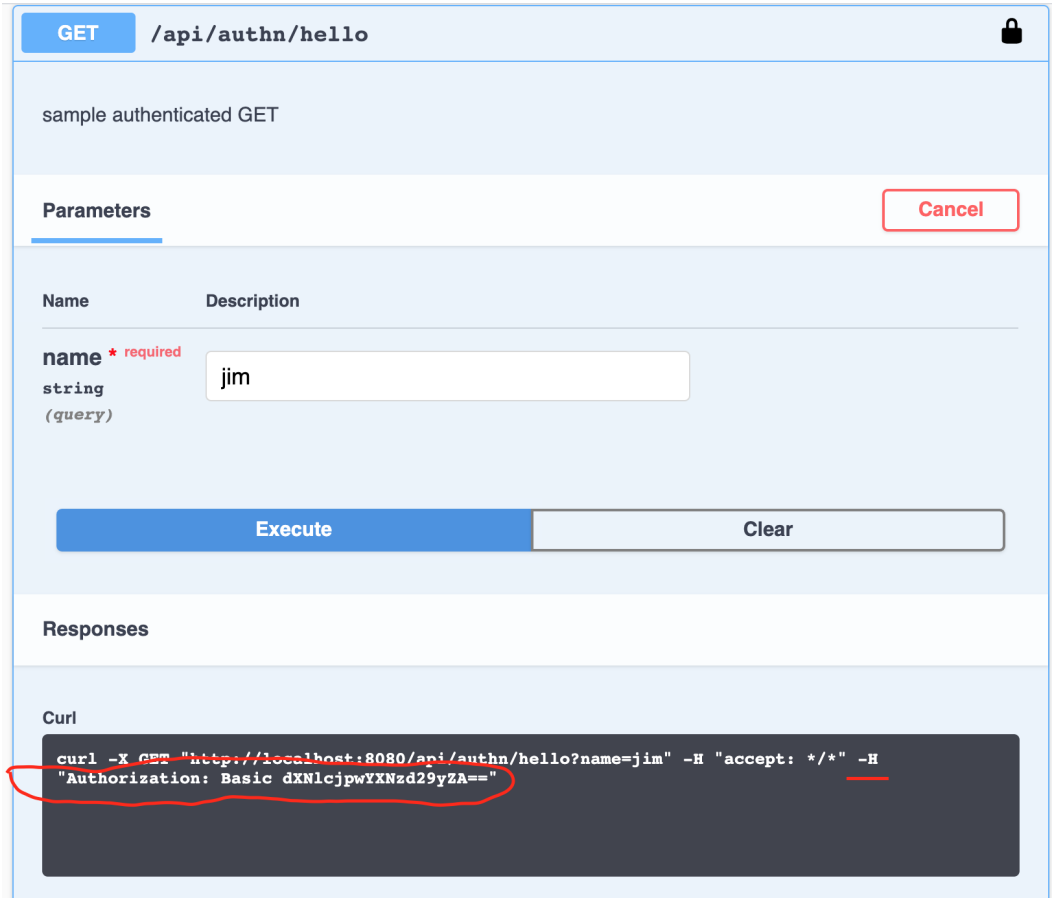


Figure 4. Swagger BASIC Auth Call

Chapter 6. CORS

There is one more important security filter to add to our list before we end, and it is complex enough to deserve its own section - Cross Origin Resource Sharing (CORS). Using this standard, browsers will supply the URL of the source of Javascript used to call the server — when coming from a different host domain — and look for a response from the server that indicates the source is approved. Without support for CORS, javascript loaded by browsers will not be able to call the API unless it was loaded from the same base URL as the API. That even includes local development (i.e., javascript loaded from file system cannot invoke `http://localhost:8080`). In today's modern web environments — it is common to deploy services independent of Javascript-based UI applications or to have the UI applications calling multiple services with different base URLs.

6.1. Default CORS Support

The following example shows the result of the default CORS configuration for Spring Boot/Web MVC. The server is ignoring the `Origin` header supplied by the client and does not return any CORS-related authorization for the browser to use the response payload.

CORS Inactive, Origin Header Ignored

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
>
< HTTP/1.1 200
hello, jim :caller=(null)

$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin:
http://127.0.0.1:8080"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Origin: http://127.0.0.1:8080 ①
>
< HTTP/1.1 200
hello, jim :caller=(null)
```

① `Origin` header normally supplied by browser when coming from a different domain — ignored by server

The lack of headers does not matter for curl, but the CORS response does get evaluated when executed within a browser.

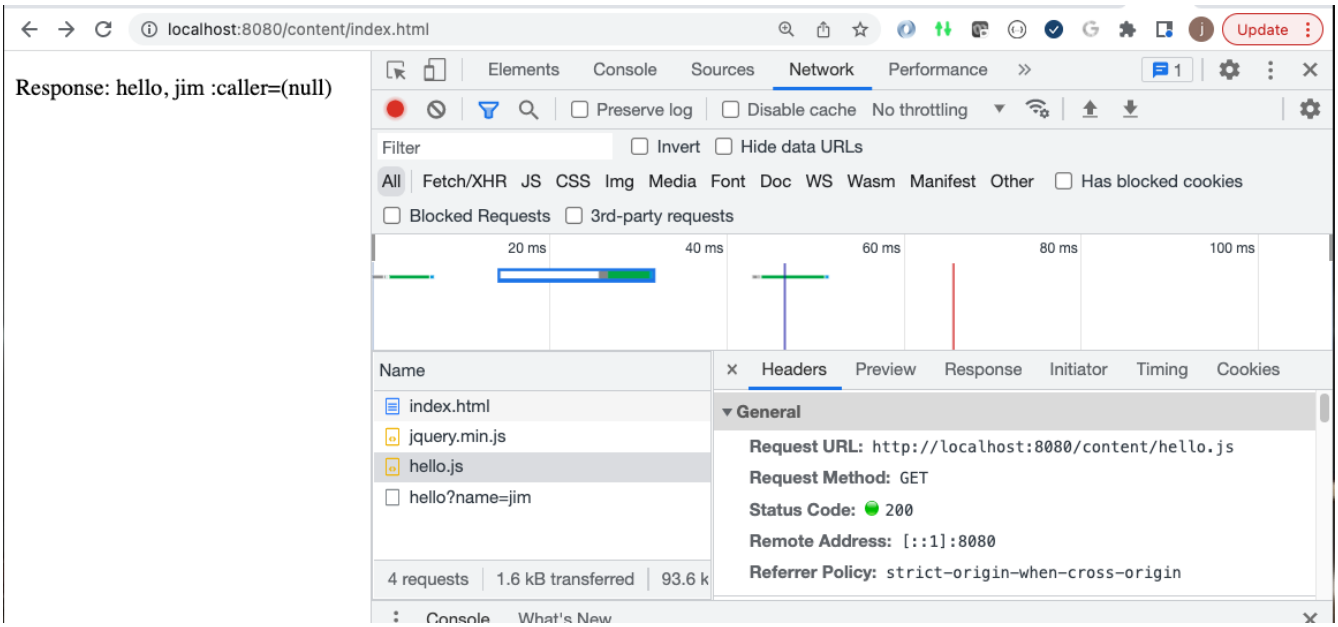
6.2. Browser and CORS Response

6.2.1. Same Origin/Target Host

The following is an example of Javascript loaded from `http://localhost:8080` and calling `http://localhost:8080`. No `Origin` header is passed by the browser because it knows the Javascript

was loaded from the same source it is calling.

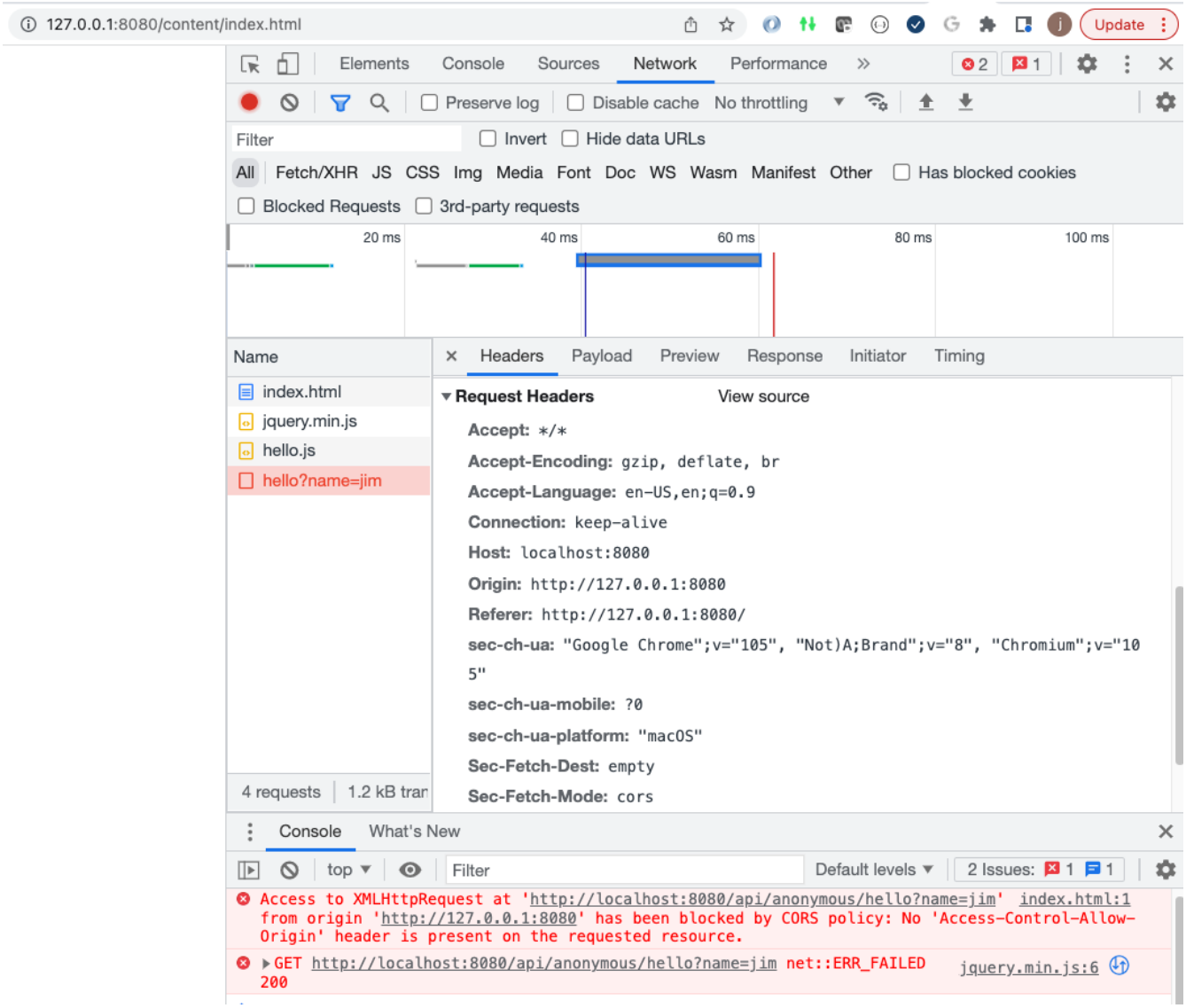
CORS Inactive, Origin Header Not Supplied for Same Source



6.2.2. Different Origin/Target Host

However, if we load the Javascript from an alternate source, the browser will fail to process the results. The following is an example of some Javascript loaded from `http://127.0.0.1:8080` and calling `http://localhost:8080`.

CORS Inactive, Origin Header Supplied for Different Source



6.3. Enabling CORS

To globally enable CORS support, we can invoke `http.cors(…)` with a method to call at runtime that will evaluate and return the result for the CORS request — based on a given `HttpServletRequest`. This is supplied when configuring the `SecurityFilterChain`.

Enabling CORS and Permit All

```
http.cors(cfg->cfg.configurationSource(corsPermitAllConfigurationSource()));
```

Example CORS Permit All Lambda Method Response

```
private CorsConfigurationSource corsPermitAllConfigurationSource() {
    return (request) -> {
        CorsConfiguration config = new CorsConfiguration();
        config.applyPermitDefaultValues();
        return config;
    };
}
```

CORS Evaluation Interface Implemented

```
package org.springframework.web.cors;

public interface CorsConfigurationSource {
    CorsConfiguration getCorsConfiguration(HttpServletRequest request);
}
```

6.3.1. CORS Headers

With CORS enabled and permitting all, we see some new VARY headers (indicating to caches that content will be influenced by these headers). The browser will be looking for the `Access-Control-Allow-Origin` header being returned with a value matching the `Origin` header passed in (* being a wildcard match).

CORS Approval Headers Returned in call cases

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080 ①
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers ②
hello, jim :caller=(null)

$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin:
http://127.0.0.1:8080"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Origin: http://127.0.0.1:8080 ③
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Access-Control-Allow-Origin: * ④
hello, jim :caller=(null)
```

① `Origin` header not supplied

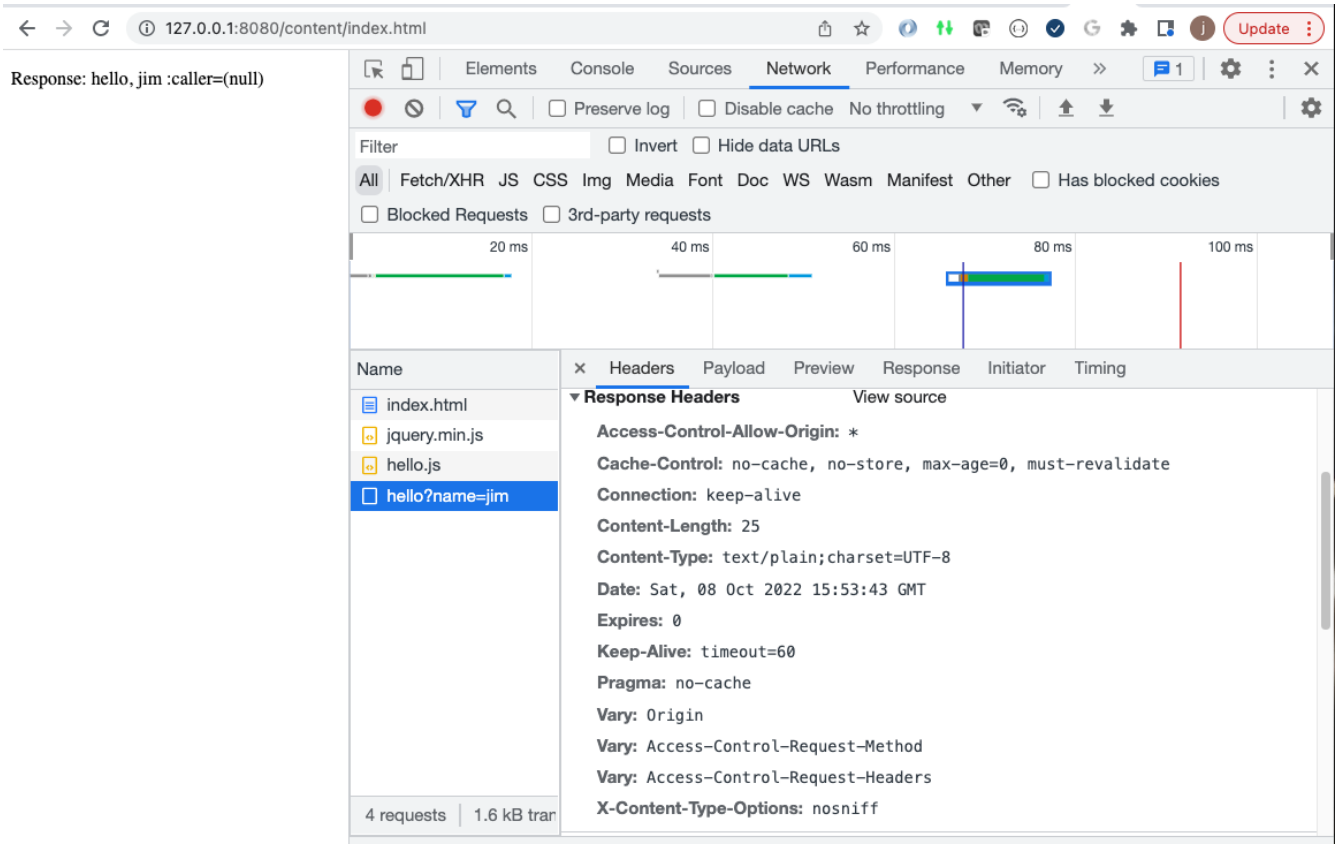
② No CORS `Access-Control-Allow-Origin` supplied in response

③ `Origin` header supplied by client

④ `Access-Control-Allow-Origin` denotes approval for the given Origin (* = wildcard)

6.3.2. Browser Accepts Access-Control-Allow-Origin Header

Browser Accepts CORS Access-Control-Allow-Origin Response



6.4. Constrained CORS

We can define more limited rules for CORS acceptance by using additional commands of the `CorsConfiguration` object.

Limiting CORS Acceptance

```
private CorsConfigurationSource corsLimitedConfigurationSource() {
    return (request) -> {
            CorsConfiguration config = new CorsConfiguration();
            config.addAllowedOrigin("http://localhost:8080");
            config.setAllowedMethods(List.of("GET","POST"));
            return config;
    };
}
```

6.5. CORS Server Acceptance

In this example, I have loaded the Javascript from `http://127.0.0.1:8080` and making a call to `http://localhost:8080` in order to match the configured `Origin` matching rules. The server is return a `200/OK` along with a `Access-Control-Allow-Origin` value that matches the specific `Origin` provided.

CORS Acceptance Response

```
$ curl -v http://127.0.0.1:8080/api/anonymous/hello?name=jim -H "Origin:
http://localhost:8080"
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: 127.0.0.1:8080 ①
> Origin: http://localhost:8080 ②
>
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Access-Control-Allow-Origin: http://localhost:8080 ②
hello, jim :caller=(null)
```

① Example Host and Origin have been flipped to match approved localhost:8080 Origin

② `Access-Control-Allow-Origin` denotes approval for the given Origin

6.6. CORS Server Rejection

This additional definition is enough to produce a `403/FORBIDDEN` from the server versus a rejection from the browser.

CORS Rejection Response

```
$ curl -v http://localhost:8080/api/anonymous/hello?name=jim -H "Origin:
http://127.0.0.1:8080"
> GET /api/anonymous/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Origin: http://127.0.0.1:8080
>
< HTTP/1.1 403
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
Invalid CORS request
```

6.7. Spring MVC @CrossOrigin Annotation

Spring also offers an annotation-based way to enable the CORS protocol. In the example below, `@CrossOrigin` annotation has been added to the controller class or individual operations indicating CORS constraints.

This technique is static.

Spring MVC @CrossOrigin Annotation

```
...
import org.springframework.web.bind.annotation.CrossOrigin;
...
@CrossOrigin ①
@RestController
public class HelloController {
```

① defaults to all origins, etc.

Chapter 7. RestTemplate Authentication

Now that we have locked down our endpoints — requiring authentication — I want to briefly show how we can authenticate with `RestTemplate` using an existing BASIC Authentication filter. I am going to delay demonstrating `WebClient` to limit the dependencies on the current example application — but we will do so in a similar way that does not change the interface to the caller.

7.1. ClientHttpRequestFactory

We will first define a factory for creating client connections. It is quite simple here because we are not addressing things like HTTP/TLS connections. However, creating the bean external from the clients makes the clients connection agnostic.

ClientHttpRequestFactory

```
@Bean
ClientHttpRequestFactory requestFactory() {
    return new SimpleClientHttpRequestFactory();
}
```

This simple `ClientRequestFactory` will get slightly more complicated when we enable SSL connections. By instantiating it now in a separate method we will make the rest of the `RestTemplate` configuration oblivious to the SSL/non-SSL configuration.

7.2. Anonymous RestTemplate

The following snippet is an example of a RestTemplate representing an anonymous user. This should look familiar to what we have used prior to security.

RestTemplate Anonymous Client

```
@Bean
public RestTemplate anonymousUser(RestTemplateBuilder builder,
                                  ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
            //used to read the streams twice -- so we can use the logging filter below
            ()->new BufferingClientHttpRequestFactory(requestFactory))
            .interceptors(new RestTemplateLoggingFilter())
            .build(); ①
    return restTemplate;
}
```

① vanilla RestTemplate with our debug log interceptor

7.3. Authenticated RestTemplate

The following snippet is an example of a RestTemplate that will authenticate as "user/password" using Http BASIC Authentication. The authentication is added as a filter along with the logger. The business code using this client will be ignorant of the extra authentication details.

RestTemplate Authenticating Client

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                              ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
            //used to read the streams twice -- so we can use the logging filter below
            ()->new BufferingClientHttpRequestFactory(requestFactory))
            .interceptors(new BasicAuthenticationInterceptor("user", "password"), ①
                    new RestTemplateLoggingFilter())
            .build();
    return restTemplate;
}
```

① added BASIC Auth filter to add Authorization Header

7.4. Authentication Integration Tests with RestTemplate

The following shows the different `RestTemplate` instances being injected that have different credentials assigned. The different attribute names, matching the `@Bean` factory names act as a qualifier to supply the right instance of `RestTemplate`.

```
@SpringBootTest(classes= ClientTestConfiguration.class,
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
        properties = "test=true") ①
public class AuthnRestTemplateNTest {
    @Autowired
    private RestTemplate anonymousUser;
    @Autowired
    private RestTemplate authnUser;
```

① `test` property triggers Swagger `@Configuration` and anything else not suitable during testing to disable

Chapter 8. RestClient Authentication

Let's also show how to authenticate with `RestClient`. We can do that using a builder or use a fully configured `RestTemplate`.

8.1. Anonymous RestClient

The following snippet is an example of a RestClient representing an anonymous user.

RestClient Anonymous Client

```
@Bean
public RestClient anonymousUserClient(RestClient.Builder builder,
ClientHttpRequestFactory requestFactory) {
    return builder.requestFactory(//used to read streams twice -- to use logging
filter
                    new BufferingClientHttpRequestFactory(requestFactory))
            .requestInterceptor(new RestTemplateLoggingFilter())
            .build();
}
```

① vanilla RestClient with our debug log interceptor

8.2. Authenticated RestTemplate

The following snippet is an example of a RestClient built from an existing RestTemplate. It will also authenticate as "user/password" using Http BASIC Authentication.

RestClient Authenticating Client

```
@Bean
public RestClient authnUserClient(RestTemplate authnUser) {
    return RestClient.create(authnUser); ①
}
```

① uses already assembled filters from RestTemplate

Chapter 9. Mock MVC Authentication

There are many test frameworks within Spring and Spring Boot that I did not cover earlier. I limited them because covering them all early on added limited value with a lot of volume. However, I do want to show you a small example of MockMvc and how it to can be configured for authentication. The following example shows a:

- normal injection of the mock that will be an anonymous user
- how to associate a mock to the security context

MockMvc Authentication Setup

```
@SpringBootTest(
        properties = "test=true")
@AutoConfigureMockMvc
public class AuthConfigMockMvcNTest {
    @Autowired
    private WebApplicationContext context;
    @Autowired
    private MockMvc anonymous; //letting MockMvc do the setup work
    private MockMvc userMock; //example manual instantiation ①
    private final String uri = "/api/anonymous/hello";

    @BeforeEach
    public void init() {
        userMock = MockMvcBuilders //the rest of manual instantiation
            .webAppContextSetup(context)
            .apply(SecurityMockMvcConfigurers.springSecurity())
            .build();
    }
```

① there is no functional difference between the injected or manually instantiated `MockMvc` the way it is performed here

9.1. MockMvc Anonymous Call

The first test is a baseline example showing a call through the mock to a service that allows all callers and no required authentication. The name of the mock is not important. It is a anonymous client at this point because we have not assigned it any identity.

MockMvc Anonymous Call

```
@Test
public void anonymous_can_call_get() throws Exception {
    anonymous.perform(MockMvcRequestBuilders.get(uri).queryParam("name","jim"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().string("hello, jim :caller=(null)"));
```

```
}
```

9.2. MockMvc Authenticated Call

The next example shows how we can inject an identity into the mock for use during the test method. We can use an injected or manual mock for this. The important point to notice is that the mock user's identity is assigned through an annotation on the `@Test`.

MockMvc Authenticated Call

```
@WithMockUser("user")
@Test
public void user_can_call_get() throws Exception {
    userMock.perform(MockMvcRequestBuilders.get(uri)
            .queryParam("name","jim"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().string("hello, jim :caller=user"));
}
```

Although I believe RestTemplate tests are pretty good at testing client access — the WebMvc framework was a very convenient to quickly verify and identify issues with the `SecurityFilterChain` definitions.

9.3. MockMvc does not require SpringBootTest

The MockMvc web test framework does not require the full application context implemented by `SpringBootTest`. MockMvc provides a means to instantiate small unit tests incorporating mocks behind the controllers. For example, I have used it as a lightweight way to test `ControllerAdvice` /`ExceptionAdvice`.

Chapter 10. Summary

In this module, we learned:

- how to configure a `SecurityFilterChain`
- how to define no security filters for static resources
- how to customize the `SecurityFilterChain` for API endpoints
- how to expose endpoints that can be called from anonymous users
- how to require authenticated users for certain endpoints
- how to CORS-enable the API
- how to define BASIC Auth for OpenAPI and for use by Swagger
- how to add identity to RestTemplate and RestClient clients