Authorization

Theory

30 minutes reading

Verify to skip Start practicing

In simple applications, authentication might be enough: as soon as a user authenticates themselves (confirms their identity), they can access any part of the application. However, in some situations, not all authenticated users should be granted access to some app resources. **Authorization** is the process during which the system decides if an authenticated client has permission to access the requested resource. Authorization always happens after authentication.

In this topic, you'll learn how to configure authorization in Spring Security. We will create a program with a couple of endpoints, configure access to them and then test the program using Postman.

1 required topic

Authentication

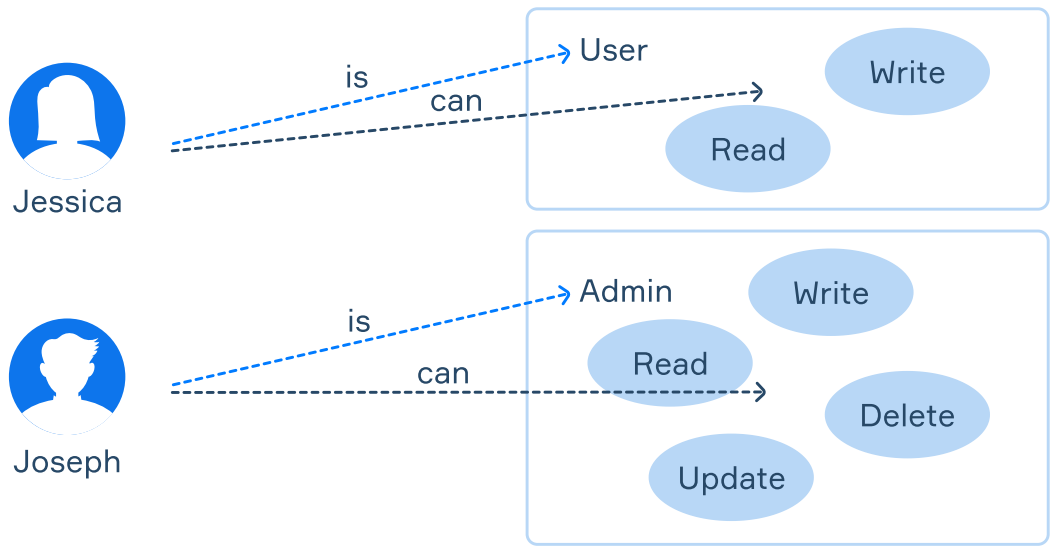
2 dependent topics

Spring actuator

Custom User Store

§1. Roles and authorities

To allow one authenticated user to access a resource and restrict access to another user we need a mechanism for distinguishing users. Spring Security provides **authorities** and **roles** to help you with this task.



- Authorities are actions that users can perform in an application. Only users with specific authorities can make a particular request to an endpoint. For example, Jessica can only `READ` and `WRITE` to the endpoint, while Joseph can `READ`, `WRITE`, `DELETE`, and `UPDATE` the endpoint. Under the hood, authority is just `String`. We can choose any name for authority when developing an app.
- Roles are groups of authorities. Imagine you're going to have two types of users in your application. One type should only be able to read and write data, while another one should be able to read, write, update, and delete it. Instead of using four authorities, we can simply define two roles. For example, `ROLE_USER` and `ROLE_ADMIN`. Under the hood, a role is `String` prefixed with `ROLE_`.

In Spring Security, the concepts of authority and role are often used interchangeably. In most cases, the difference is just in the naming convention used. It will become more clear once you've seen some examples.

In the program that we are going to write, we will configure access to endpoints based on user roles. Our program will have the following functionality:

- `/`, `/public` available without authentication (no login/password is required).
- `/authenticated` available only to authenticated users.
- `/user` available only to authenticated users with a `ROLE_USER` or `ROLE_ADMIN` role.
- `/admin` available only to authenticated users with a `ROLE_ADMIN` role.

We will start implementing the program by adding a controller and creating users with roles. After that, we will deal with authorization.

§2. Initial setup

Let's assume that we've already created a new empty project and included web and security dependency. Here is our controller:

▼ Java

```java
import org.springframework.web.bind.annotation.*;

@RestController
public class Controller {

    @GetMapping("/public")
    public String testPublic() {
        return "/public is accessed";
    }


    @GetMapping("/authenticated")
    public String testAuth() {
        return "/authenticated is accessed";
    }


    @GetMapping("/user")
    public String testUser() {
        return "/user is accessed";
    }


    @GetMapping("/admin")
    public String testAdmin() {
        return "/admin is accessed";
    }
}
```

▼ Kotlin

```kotlin
import org.springframework.web.bind.annotation.*

@RestController
class Controller {

    @GetMapping("/public")
    fun testPublic(): String {
```

```
        return "/public is accessed"
    }


    @GetMapping("/authenticated")
    fun testAuth(): String {
        return "/authenticated is accessed"
    }


    @GetMapping("/user")
    fun testUser(): String {
        return "/user is accessed"
    }


    @GetMapping("/admin")
    fun testAdmin(): String {
        return "/admin is accessed"
    }
}
```

Now, let's create users and assign roles. We'll need 3 users: a user without a role, a user with `ROLE_USER`, and a user with `ROLE_ADMIN`. For simplicity's sake, we will create hardcoded users in memory, but overall it doesn't matter whether the users are stored in memory, in a database, or somewhere else. The process of configuring authorization is similar.

▼ Java

To assign a role to a user, we'll use the `roles(String... roles)` method that receives zero or more roles. There is a similar method for authorities `authorities(String... authorities)`, but we are not going to use authorities in this topic.

▼ Kotlin

To assign a role to a user, we'll use the `roles(vararg roles: String)` method that receives zero or more roles. There is a similar method for authorities `authorities(vararg authorities: String)`, but we are not going to use authorities in this topic.

Here is what our implementation looks like:

▼ Java

```
// imports ..

@EnableWebSecurity
public class WebSecurityConfigurerImpl extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
        auth.inMemoryAuthentication()
```

```
.withUser("user1").password(getEncoder().encode("pass1")).roles()
            .and()
.withUser("user2").password(getEncoder().encode("pass2")).roles("USER")
            .and()
.withUser("user3").password(getEncoder().encode("pass3")).roles("ADMIN")
            .and()
            .passwordEncoder(getEncoder());
    }

    @Bean
    public PasswordEncoder getEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

▼ Kotlin

```
// imports ..

@EnableWebSecurity
class WebSecurityConfigurerImpl : WebSecurityConfigurerAdapter() {

    override fun configure(auth: AuthenticationManagerBuilder) {
        auth.inMemoryAuthentication()
.withUser("user1").password(getEncoder().encode("pass1")).roles()
            .and()
.withUser("user2").password(getEncoder().encode("pass2")).roles("USER")
            .and()
.withUser("user3").password(getEncoder().encode("pass3")).roles("ADMIN")
            .and()
            .passwordEncoder(getEncoder())
    }

    @Bean
    fun getEncoder() = BCryptPasswordEncoder()
}
```

Note that we don't have to add the `ROLE_` prefix to roles. Spring Security will add it automatically. We can also use the `authorities()` method to specify a role, but in this case, it must be prefixed with `ROLE_`. Other than that,

`roles("ADMIN")` is equivalent to `authorities("ROLE_ADMIN")`. This knowledge can help you avoid some mistakes and confusion.

Our program will also have a simple `index.html` file located in the `/resources/static` folder. Here's the content of the file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Public</title>
</head>
<body>
    <h1>Welcome stranger!</h1>
</body>

</html>
```

Now, let's deal with authorization. We will start by learning about some useful methods.

§3. HttpSecurity

We can configure authorization using the `HttpSecurity` object which can be obtained by overriding one of the `configure` methods of `WebSecurityConfigurerAdapter`. This is the same object that allows us to configure form-based and HTTP basic auth.

In this topic, we are interested in some of the methods that we can call using this object. The methods in question can be divided into two groups. One group allows specifying endpoints to which we want to configure access and another group allows specifying the users who can access these endpoints.

Using the following methods we can select the endpoints:

▼ Java

- `mvcMatchers(HttpMethod method, String... patterns)` lets us specify both the HTTP method to which the restrictions apply and the paths. This method is useful if you want to apply different restrictions to different HTTP methods with the same path. Here's an example of how you can select a `GET /public` endpoint: `mvcMatchers(HttpMethod.GET, "/public")`.
- `mvcMatchers(String... patterns)` is similar to the previous method but only allows specifying paths. The restrictions automatically apply to any HTTP method.
- `regexMatchers(HttpMethod method, String... regex)` and `regexMatchers(String... regex)` are similar to the MVC matchers but allow using regular expressions.
- `anyRequest()` maps any request, regardless of the URL or HTTP method used.

▼ Kotlin

- `mvcMatchers(method: HttpMethod, vararg patterns: String)` lets us specify both the HTTP method to which the restrictions apply and the paths. This method is useful if you want to apply different restrictions to different HTTP methods with the same path. Here's an example of how you can select a `GET /public` endpoint: `mvcMatchers(HttpMethod.GET, "/public")`.
- `mvcMatchers(vararg patterns: String)` is similar to the previous method but only allows specifying paths. The restrictions automatically apply to any HTTP method.
- `regexMatchers(method: HttpMethod, vararg regex: String)` and `regexMatchers(vararg regex: String)` are similar to the MVC matchers but allow using regular expressions.

- `anyRequest()` maps any request, regardless of the URL or HTTP method used.

> Spring Security also provides Ant matchers that are similar to the MVC matchers mentioned above. The difference is that the Ant matchers match only exact URLs. For example, `antMatchers("/secured")` will match only `/secured`, while `mvcMatchers("/secured")` will match `/secured`, as well as `/secured/` and `/secured.html`. It is recommended to use the MVC matchers to avoid situations where some paths are mistakenly left unprotected.

The following methods allow us to specify who can access endpoints:

▼ Java

- `permitAll()` specifies that anyone can access a URL (authenticated and not authenticated users).
- `denyAll()` specifies that no one can access a URL.
- `authenticated()` specifies that any authenticated user can access a URL.
- `hasRole(String role)` is a shortcut for specifying URLs requiring a particular role. It should not start with `ROLE_` as it is automatically inserted.
- `hasAnyRole(String... roles)` is the same as the previous method but allows specifying multiple roles.
- `hasAuthority(String authority)` specifies that access to a URL requires a particular authority. We can also use this method to specify a role but it should be prefixed with `ROLE_`. That is, `hasAuthority("ROLE_ADMIN")` is similar to `hasRole("ADMIN")`.
- `hasAnyAuthority(String... authorities)` is the same as the previous method but allows specifying multiple authorities (or roles).

▼ Kotlin

- `permitAll()` specifies that anyone can access a URL (authenticated and not authenticated users).
- `denyAll()` specifies that no one can access a URL.
- `authenticated()` specifies that any authenticated user can access a URL.
- `hasRole(role: String)` is a shortcut for specifying URLs requiring a particular role. It should not start with `ROLE_` as it is automatically inserted.
- `hasAnyRole(vararg roles: String)` is the same as the previous method but allows specifying multiple roles.
- `hasAuthority(authority: String)` specifies that access to a URL requires a particular authority. We can also use this method to specify a role but it should be prefixed with `ROLE_`. That is, `hasAuthority("ROLE_ADMIN")` is similar to `hasRole("ADMIN")`.
- `hasAnyAuthority(vararg authorities: String)` is the same as the previous method but allows specifying multiple authorities (or roles).

The MVC matchers and the Ant matchers support **wildcards**:

- `?` matches one character. For example, `mvcMatchers("/?")` will match `/a` and `/b` but not `/` or `/ab`.
- `*` matches zero or more characters. For example, `mvcMatchers("/*")` will match `/`, `/a`, and `/ab` but not `/a/b`.
- `**` matches zero or more directories in a path. For example, `mvcMatchers("/**")` will match `/`, `/a`, `/ab`, and `/a/b/c`.

Wildcards can be placed at any point of a path. Here are some examples: `/page/?`, `/page/*/comments`, `/api/**`.

Now let's put the pieces together and finish our program.

§4. Configuring authorization

First, we override the `configure` method that receives the `HttpSecurity` object and call the `authorizeRequests()` method:

▼ Java

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    // more methods
}
```

▼ Kotlin

```kotlin
override fun configure(http: HttpSecurity) {
    http.authorizeRequests()
    // more methods
}
```

To configure access to our endpoints, we stick to the following rule: first, append a call to a method that allows for selecting endpoints. After that, append a call to a method that allows for specifying who can access endpoints. Here is how we can make `/admin` available only to users with the `ROLE_ADMIN` role:

```
 .mvcMatchers("/admin").hasRole("ADMIN") // or
.hasAuthority("ROLE_ADMIN")
```

If we need more rules, we can append more method pairs at the end of the method chain. Here is the full example:

▼ Java

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .mvcMatchers("/admin").hasRole("ADMIN")
            .mvcMatchers("/user").hasAnyRole("ADMIN", "USER")
            .mvcMatchers("/", "/public").permitAll()
            .mvcMatchers("/**").authenticated() // or
.anyRequest().authenticated()
            .and().httpBasic();
}
```

▼ Kotlin

```kotlin
override fun configure(http: HttpSecurity) {
    http.authorizeRequests()
        .mvcMatchers("/admin").hasRole("ADMIN")
        .mvcMatchers("/user").hasAnyRole("ADMIN", "USER")
        .mvcMatchers("/", "/public").permitAll()
        .mvcMatchers("/**").authenticated() // or
.anyRequest().authenticated()
        .and().httpBasic()
}
```

For learning purposes, we didn't specify `/authenticated` explicitly and instead used `/**` which selects all URLs including `/authenticated`.

Note that we've placed `.mvcMatchers("/**").authenticated()` at the end. That's because the order of methods is important. Methods are considered in the order they were declared. If we place `.mvcMatchers("/**").authenticated()` right after `.authorizeRequests()`, the whole application will be available only to authenticated users, and the remaining rules will be ignored. Now imagine that we have a real program and instead of `.authenticated()` we specified `.permitAll()`

and placed it at the top. The whole application will be available to everyone. Pay attention when configuring authorization. The order of rules must be from specific to general.

Keep in mind that we use the MVC matchers here. `.mvcMatchers("/admin")` additionally matches `"/admin/"` that will also be available only to users with the `ROLE_ADMIN` role. But what will happen if we replace an MVC matcher with an Ant matcher that matches only exact URLs? `/admin` will require the `ROLE_ADMIN` role as intended, but `/admin/` will only require authentication (`/**` represents all the remaining paths in our case). This makes a huge difference!

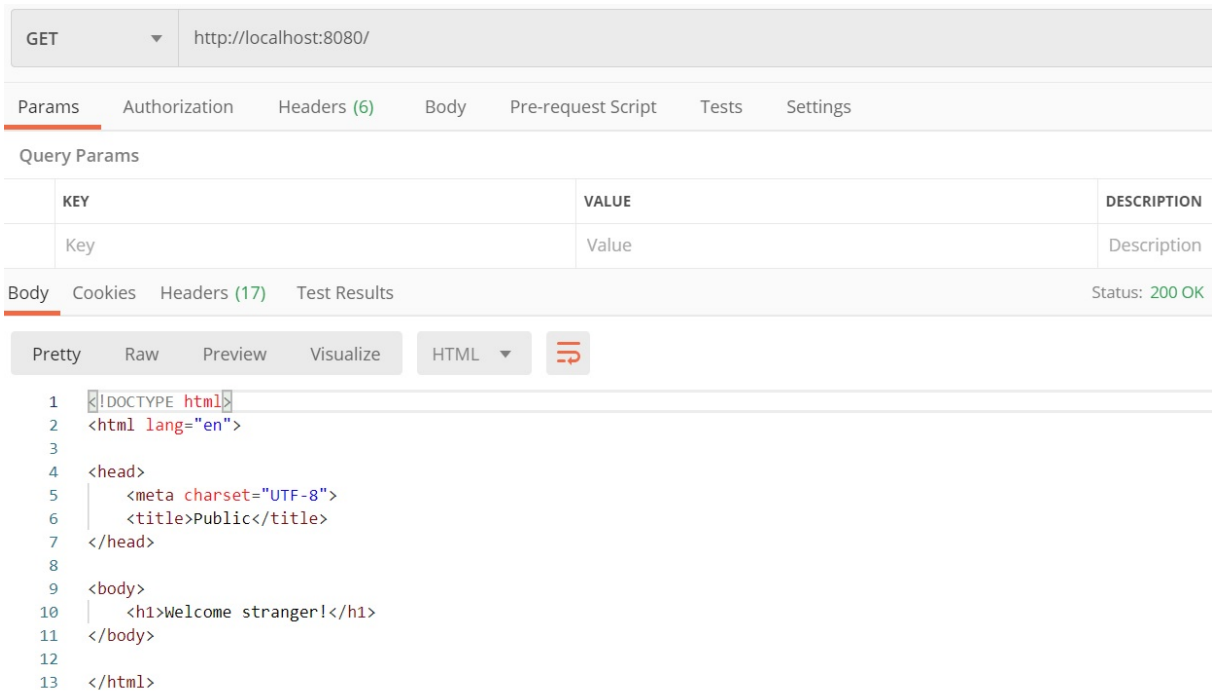
A developer who is unaware of this could use the Ant matchers and leave a path unprotected without noticing it, which can create a major security breach in an application. It doesn't mean we should never use the Ant matchers, though. We can still secure two paths using the Ant matchers like this: `antMatchers("/admin", "/admin/")`.

Another way to configure authorization is by using special annotations. We will not consider it in this topic, but you can learn more about them in an article (Java).

As you can see, we've also enabled HTTP basic authentication. Now we are going to use this type of auth and Postman to test our program.

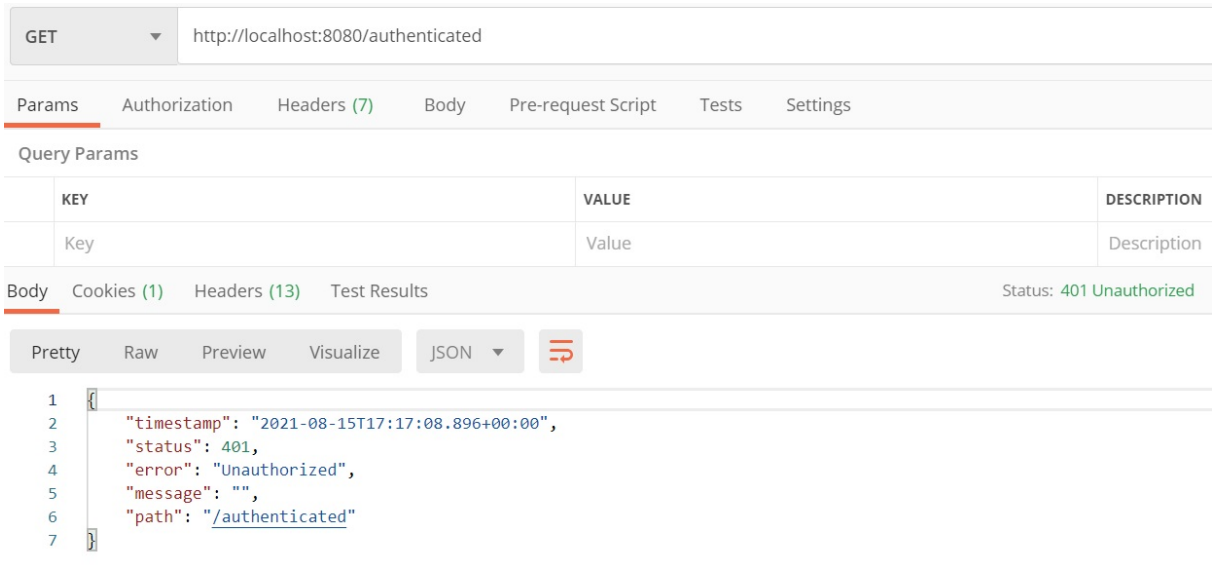
§5. Running the app

Let's run the program and then try to access `/`, which is supposed to be public:

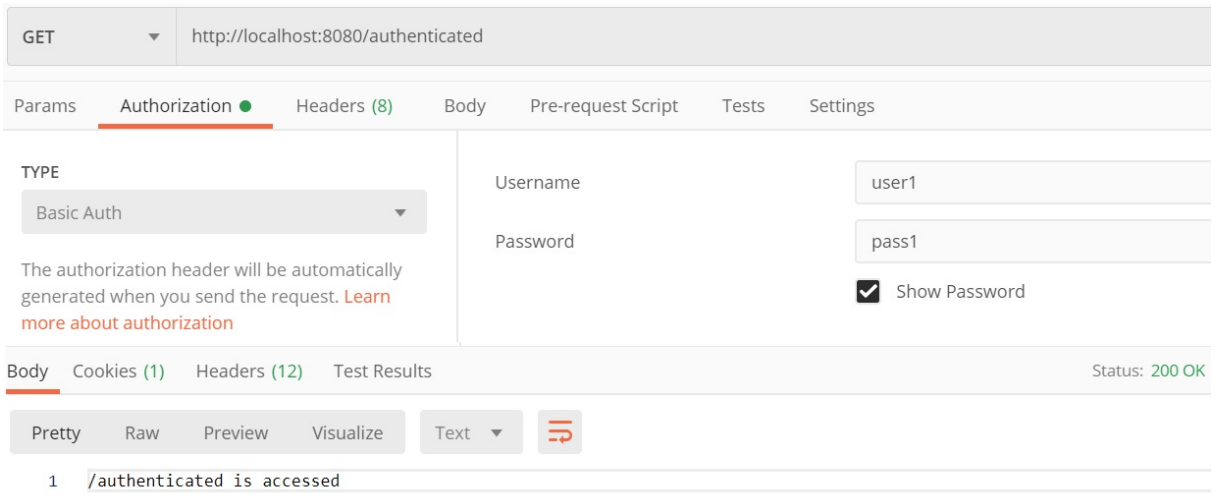


As expected the status code is `200 OK` and we see the content of `index.html`. You can try to access `/public`, it will also work.

Now if we try to access the remaining URLs in the same way we shouldn't be able to do that because the remaining URLs require at least authentication. Here's an attempt to access `/authenticated` (or `/user` `/admin`):

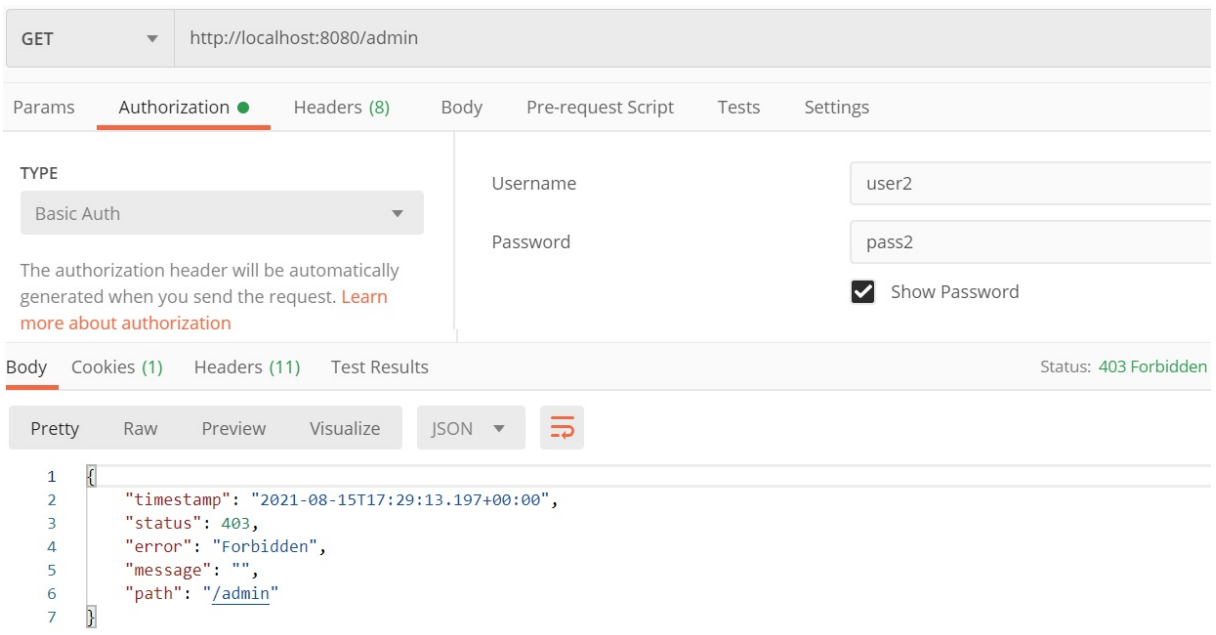


As we can see, the request couldn't be satisfied and we've received a `401 Unauthorized` status code that indicates that the request requires user authentication. To fix the error, we need to use HTTP basic auth and input one of the valid login/password pairs. This can be done in the "Authorization" section of Postman. Let's use the login and password of the first user that doesn't have a role and try again.



Note that apart from login and password we also specified the authentication type, making it "Basic Auth". As you can see, we were able to access our program and the status code is `200 OK`. If we try to access `/authenticated` using the two remaining users the result will be the same.

Now what will happen if we try to access `/user` or `/admin` as a user that doesn't have the required role? Let's try to access `/admin` as a user with the `ROLE_USER` role:



Even though we've provided one of the valid login/password pairs and passed authentication, we've received `403 Forbidden` status, which means that the server understood the request but refused to authorize it. The same will happen if we try to access it as the first user. Only users with `ROLE_ADMIN` are authorized to access `/admin` (or `/admin/`). So our application works as intended.

> As you probably know, CSRF (Cross-Site Request Forgery) is enabled by default. If we try to send `POST` requests using Postman or similar programs, we will receive the `403 Forbidden` status code because of this protection. In our program, we only have `GET` endpoints so we didn't have any issues. When testing `POST` requests, you can disable this type of protection by calling the `.csrf().disable()` methods on the `HttpSecurity` object. Depending on your implementation you may also need to append a call to `.and()` before the methods.

Feel free to experiment with this. You may also want to enable form-based auth and try to access endpoints using a browser. The program will behave similarly.

§6. Conclusion

Table of contents:

In this topic, you've seen how authentication and authorization work together to make programs more secure. You've seen how we can make endpoints public or available only to authenticated users, or users with a specific role/authority.

Even though the program that we've created is just an example, it is quite representative of real programs. For example, registration and the home page of the platform are public and anyone can access them. Also, to access the main contents of the platform a user needs to pass authentication (after registration). There are also functionalities available only to the admins of the platform and hidden from regular users. For example, an admin can edit and delete comments.

Be careful when developing/maintaining programs that contain sensitive data. It's easy to make a mistake and leave some parts of your program unprotected. Programs that contain sensitive data should be properly tested.

Report a typo

120 users liked this piece of theory. 5 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Verify to skip

Comments (11) Useful links (2) Show discussion