Exercises for Advanced Topics of Software Engineering
Winter semester 2021/22
Prof. Dr. Florian Matthes, Tri Huynh, Burak Öz

Technische Universität München
Fakultät für Informatik

Exercise Sheet 6: Safety Part II and Authentication using Spring

Welcome to the sixth week of Advanced Topics of Software Engineering. This week, we will focus on the remaining topics on safety and work on building an authentication system. First, we are going to conduct a fault tree analysis for a failing braking system. Later, we will practice how to create an authentication system backend with Spring.

Theoretical Exercise

Exercise 1: Fault Tree Analysis

Fault Tree Analysis (FTA) is a graphical representation of causal relationships that aims to identify all conditions that lead to a system failure (top-level event) using a deductive top-down approach. It is an ideal way to perform a quantitative analysis.

You are working for CampusShare, the company that provides traveling vehicles to university campuses located in Florida, USA. To raise awareness about the decreasing population of American alligators, the company recently introduced a new e-scooter called *Gator*. Gator is expected to be very popular among students as it offers an autonomous driver option to several points on the campus.

Although Gator attracted many students on the first week of the new semester, the university board decided to ban the use of Gators due to the increasing number of accidents. The accident reports show that the root cause of the accidents is the failing braking system of the Gator's autonomous driver.

Your superior asked you to perform an FTA on the braking system of Gator's autonomous driver mode to identify conditions which cause a failure. From previous accident reports and interviews with the technicians, you have obtained the following information:

- The braking system of Gator's autonomous driver mode consists of three embedded brakes (Components B1, B2, B3) and a braking controller (Component C1).
- The complete braking system fails if either two of the three breaks fail or the braking controller overheats and shuts down, which can occur often due to hot Florida weather.
- While the individual availability of all three brakes is the same with $a_{b1} = a_{b2} = a_{b3} = 0.95$, the probability of the braking controller overheating is $a_{c1} = 0.3$.

Your next tasks are thus as follows:

1. Draw the fault tree with respect to the above mentioned information. Start with the root event $brakingsystem_{fails}$, then continue with intermediate events and the basic event of braking controller overheating, and then decompose the intermediate events as much as possible.
2. Calculate the probability that the root event, $brakingsystem_{fails}$, occurs.
3. Calculate the minimal cut-sets that lead to a failure of the braking system.

Practical Exercise

In the following, you will get to know the framework **Spring Security**, which is designed to cope with these challenges and is one of numerous libraries and frameworks facilitating the implementation of proper authentication and authorization patterns.

This week aims to provide a solid understanding of the framework and best practices to prevent common exploits. The take away for you of this exercise sheet is to understand how to establish a robust mechanism to ascertain, that these authorization rules are enforced.

What we will cover this week: Basic Concepts and Terminology

Concretely, you will have a closer look at the implementation aspects of **authentication** and **authorization**. These terms – although often (mistakenly) used interchangeably – describe two distinct concepts in security design:

- **Authentication** refers to the process of determining that someone is really who they claim to be.
- **Authorization** pertains to rules that determine who is allowed to do what, i.e. access control.

In the context of web application development, the authentication process is often roughly described as validating user credentials (i.e. username and password), whereas authorization implements checking a user's permission to access data or perform actions. In the ASEDelivery, we want users to authenticate themselves before they are authorized to see specific sensitive data, which in our case are users' data and deliveries information.

Be aware that understanding these concepts is crucial to anyone developing (to a certain degree) *secure* software, even though unfortunately robust solutions are often neglected due to their complexity.

By the end of these exercises you will have an understanding of how to secure your Spring Boot applications in terms of user authentication and how to define and enforce versatile access-control rules.

Introduction to Spring Security

Spring Security is another open-source Java library developed by the Spring team, which provides support for **authentication**, **authorization**, and **protection** against common exploits. The documentation is available at https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/ and discusses also how to use Spring Security in Spring Boot applications[1]. The plethora of configuration options can be intimidating at first. Therefore, we provide a conceptual introduction and a working implementation example.

Note: This introduces you to the inner process of a very mature and battle-proof software framework. Reading and understanding the API reference will give you an idea on how modern modular software development is done at a professional level. Don't let this chance slip!

Important: It is **not necessary** to understand Spring Security in depth to use it in practice. Using Spring Security not 100% correctly is not as secure as intended by the creators of the library, but still more secure than implementing security protocols or mechanisms by yourselves.

1. **Core Components**

   Spring Security has certain central interfaces, classes and conceptual abstractions that are used throughout the framework and establish support for authentication and access-control. These core components and the processes in which they interact are introduced in the following.

   - **`Principal`**: Represents the abstract notion of a *principal*, which can be used to represent any entity, such as an individual (simply a user), a corporation, and a login id[2].
   - **`SecurityContext`**: Interface defining the minimum security information associated with the current thread of execution. Holds an `Authentication` object which represents details of the `Principal` currently interacting with the application.

[1] Spring Boot Security in short: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security
[2] Principal in Java SE: https://docs.oracle.com/javase/8/docs/api/java/security/Principal.html

- **`SecurityContextHolder`**: Stores details of the present `SecurityContext` (e.g. user details) inside a `ThreadLocal`[3], which is cleared after the request is processed.
- **`Authentication`**: Interface that represents either the token for an authentication request or an authenticated `Principal` once the request has been processed by the `AuthenticationManager` which in turn uses so-called `AuthenticationProvider`s for authenticating requests.
- **`GrantedAuthority`**: An authority that is granted to the `Principal` in an `Authentication` object. Such authorities are usually *roles*, such as `ROLE_USER` or `ROLE_ADMIN`.
- **`UserDetailsService`**: Core interface – also referred to as user data access object (DAO) – which is used to retrieve a user's authentication (i.e. credentials) and authorization (i.e. authorities) information. It has a single read-only method `loadUserByUsername(String username)` which returns a `UserDetails` object.
- **`UserDetails`**: An interface containing core user information (e.g. credentials). Think of it as the adapter between your own user database and what Spring Security needs in the `Authentication` object inside the `SecurityContextHolder`.

2. **Authentication**

   The authentication process in Spring Security illustrated in Fig. 1 can be summarized as follows:

   - A registered servlet filter (`AuthenticationFilter`[4]) processes an authentication form submission (i.e. login form with HTTP POST) and extracts user credentials
   - The credentials are combined into an instance of `UsernamePasswordAuthenticationToken`
   - The token is passed to an instance of `AuthenticationManager` for validation
   - The `AuthenticationManager` returns a fully populated `Authentication` instance on successful authentication
   - Security context is established by calling `setAuthentication()` with returned `Authentication` instance on `SecurityContextHolder.getContext()`

   Finally, you can use the following code block – from anywhere in your application – to obtain the name of the currently authenticated user:

```
// Obtaining information about the current user
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

// The principal is just an Object. Most of the time this can be cast into a UserDetails
    object.
String username;
if (principal instanceof UserDetails) {
  username = ((UserDetails)principal).getUsername();
} else {
  username = principal.toString();
}
```

3. **Authorization/Access-Control**

   The main interface responsible for making access-control decisions in Spring Security is the interface `AccessDecisionManager` with its `decide()` method. Authorization is usually performed by providing class- or method-level annotations, such as `@Pre-/@PostAuthorize` or `@Pre-/@PostFilter` which we will get to know in the application example below.

   If you are interested how the authorization works under the hood: Spring Security instruments a so-called *around advice* for method invocations using Spring's standard aspect-oriented programming (AOP) support. It achieves the around advice for web requests using a standard servlet filter[6].

Exercise 2: Spring Security Basic Authentication Example

In the following, you will apply the outlined basic concepts of authentication and authorization by implementing the login for members of the ASE Project using Spring Security.

[3] ThreadLocal in Java SE: https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html

[4] Example for authentication filter `UsernamePasswordAuthenticationFilter`: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/authentication/UsernamePasswordAuthenticationFilter.html

[5] Spring Security Authentication: http://shazsterblog.blogspot.com/2018/10/spring-security-authentication-security.html

[6] Aspect-Oriented Programming in Spring: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop

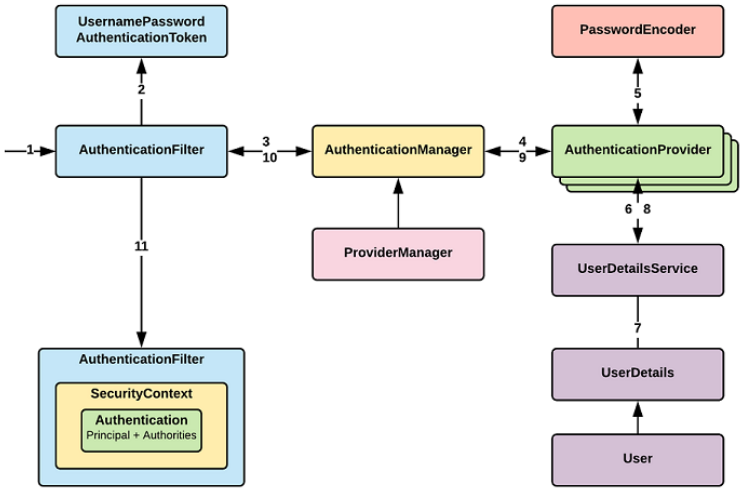

Figure 1: Spring Security Authentication Architecture[5]

- Spring Boot provides a `spring-boot-starter-security` starter (Maven dependency) that aggregates Spring Security-related dependencies. You can simply create an auto-configured app with Spring Security with the Spring Initializr (https://start.spring.io/). However, in this exercise, we will extend from the work of the previous *Project Management* example. Our missions for today are:

  1. Configure the Spring Security to authenticate members using BasicAuth
  2. Create a `AseUser` collection to store our project members and a repository to query the users in the database.
  3. Call the repository to retrieve a user from the database by leveraging a `UserDetailsService` interface provided by Spring.
  4. Create a service to receive requests and authenticate users
  5. Solve the final challenge (see it below)

Mission 1: Configure the Spring Security for Authentication

Let us start by configuring the authentication strategy for our Spring application. First, for the demo purpose of the exercise, we will use the default login form provided by Spring to receive user credentials. Spring also loads this login form when the users access resources that require authentication or authorization. Second, we will let everyone access our /auth endpoint for login, but they must be authenticated to access the resources in other endpoints. Next, our application uses the Basic Authentication[7] (e.g. content-type `applicaton/json` requested) or a login form (e.g. when the client is browser and requests `text/html`) mechanism to receive user credentials.

For the scope of this exercise, we will disable the Cross Site Request Forgery (CSRF) and Cross Origins Resource Sharing (CORS). Yet, we will implement them in the next exercise.

Create a class `SecurityConfig` to configure the Spring Security in `src/main/java/edu/ase-/project/config` by extending the `WebSecurityConfigurerAdapter`:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
    http
        .formLogin() // 1. Use Spring Login Form
        .and()
        .csrf().disable() // Note: for demonstration purposes only, this should not be done
        .authorizeRequests() // 2. Require authentication in all endpoints except login
            .antMatchers("/**").authenticated()
            .antMatchers("/auth/**").permitAll()

        .and()
        .httpBasic() // 3. Use Basic Authentication
        .and()
        .sessionManagement().disable();
```

[7]Basic Authentication is a simple authentication method, where the client provides username and password in each request as an `Authorization: Basic <encoded-credentials>` HTTP header (https://en.wikipedia.org/wiki/Basic_access_authentication)

```
    }

    @Override
    @Bean
    // Define an authentication manager to execute authentication services
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    // Define an instance of Bcrypt for hashing passwords
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

You may have sensed a code smell from the above configuration, but we will see the problem and how to solve it in the last Mission.

Run the application (IDE or `mvn spring-boot:run`) and open your browser at `localhost:8080`. You should be redirected to `localhost:8080/login` and see a login form displayed. Next, use *Postman* (discussed in previous exercises) to send an HTTP GET request to `localhost:8080`. You should receive a JSON response which looks similar to the following and a response header `WWW-Authenticate: Basic realm="Realm"`:

```
{
  "timestamp": "2021-09-28T15:49:05.230+00:00",
  "status": 401,
  "error": "Unauthorized",
  "path": "/"
}
```

What we just encountered is Spring Security's auto-configuration, more precisely the content-negotiation strategy, which decides if HTTP Basic Authentication is presented to the client.

By default, Spring secures all endpoints, which means we cannot access any parts of our web application without being authenticated. Therefore, Spring Security manages `SecurityFilterChains` which consist of Spring Security filters. You should see the `DefaultSecurityFilterChain` in your application's logs.

Mission 2: Create a `@Document` Model for Users

Before authentication, we need a collection to store our users first. Let us create a MongoDB collection for storing our `AseUser` in `src/main/java/edu/ase/project/model`. Each user will have username, password, and role. We deliberately choose not to name our model as `User`, which is used by Spring Security.

```
@Document(collection = "users")
public class AseUser {

    // TODO: Define username, password and role properties

    // Getters and Setters
}
```

For convenient role definition and access, we can create a `UserRole` class to define all the role values:

In the main `ProjectApplication`, create several users in the `CommandLineRunner` to test our authentication service later on.

```
@SpringBootApplication
@EnableMongoRepositories(basePackageClasses = {ProjectRepository.class})
public class ProjectApplication implements CommandLineRunner{

    //...

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    @Override
    public void run(String... args) throws Exception {
        log.info("MongoClient = " + mongoClient.getClusterDescription());
```

```
        // Only create test users when there is no user in DB
        if (userRepository.findAll().size() == 0) {
            // TODO: Create test users with hashed Bcrypt password and role
        }
    }
}
```

Next, create a repository `UserRepository` to query our `AseUser` Document in `src/main/java/edu/ase/project/repository`.

```
package edu.tum.ase.project.repository;

import edu.tum.ase.project.model.Project;
import edu.tum.ase.project.model.User;
import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;

// The MongoRepository is typed to the Document, and the type of the Document's ID
// TODO: Create User Repository to retrieve the User from database
```

Mission 3: Create our `UserDetailsService` to call the Repository

Spring models the core information for authenticating and authorizing users in a `User`[8] class. We will leverage this `User` to store the username, password, and role of our users.

Let us create a `UserDetailsService` in the `src/main/java/edu/ase/project/service` package to find if a user exists in our database from a given username .

```
@Component
public class MongoUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // TODO: Call the repository to find the user from a given username

        // TODO: return a Spring User with the
        // username, password and authority that we retrieved above
        return null;

    }
}
```

After you finish the implementation, register the `MongoUserDetailsService` to the Authentication Manager by first *@Autowired* into the `SecurityConfig` class, and *Override* a `configure(AuthenticationManagerBuilder)` method to specify our `MongoUserDetailsService` as the UserDetailsService to model core user information for authentication and authorization.

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    MongoUserDetailsService mongoUserDetailsService;

    @Override
    public void configure(AuthenticationManagerBuilder builder) throws Exception {
        builder.userDetailsService(mongoUserDetailsService);
    }

    // Http Config, Authentication Manager Bean Definition, and BcryptPasswordEncoder
}
```

Mission 4: Implement an Authentication Service

After retrieving the user information from the database, we can verify whether the credential from the request matches any user data in our system.

Let us create an `AuthController` to accept authentication request:

[8] Spring User: https://docs.spring.io/spring-security/site/docs/4.2.15.RELEASE/apidocs/org/springframework/security/core/userdetails/User.html

```java
package edu.tum.ase.project.controller;
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private AuthService authService;

    @PostMapping
    // TODO: Implement Authentication of the user credentials
}
```

The `@RequestHeader`[9] tag helps us to retrieve the value of any non-readonly header in the HTTP request. `HttpServletRequest` and `HttpServletResponse` represent the object of the client request, and the response that your system will return to the client.

Let us create an authentication service `AuthService` in the `src/main/java/edu/ase/project-/service` package to authenticate users.

```java
@RestController
public class AuthService {

    @Autowired
    private AuthenticationManager authManager;

    @Autowired
    private PasswordEncoder bcryptPasswordEncoder;

    @Autowired
    private MongoUserDetailsService mongoUserDetailsService;

    public ResponseEntity<String> authenticateUser(
            String authorization,
            HttpServletRequest request) throws Exception {

        // TODO: Get the username and password by decoding the Base64 credential inside
        // the Basic Authentication

        // TODO: find if there is any user exists in the database based on the credential,
        // and authenticate the user using the Spring Authentication Manager

    }
}
```

The result of a Basic Authentication is an Authorization header with a value of "Basic <username:password>" where <username:password> is in Base64 format. Extracting the username and password from the Authorization header will help us verify if any user in our database matches the credentials in the request.

After retrieving the username and password, you can use the `UserDetailsService` to load the `User` that matches the given username. Next, use the `AuthenticationManager`[10] to authenticate the user. Hint: you can use `UsernamePasswordAuthenticationToken` as the `Authentication` object.

At this point, you can log in to your browser by accessing the localhost:8080/login (the right way), or localhost:8080/project (the unintended way). The Spring login form should be displayed, and you can enter the username and password of a user you created from Mission 1 to test the authentication.

After successfully logging in, if you enter the system via the localhost:8080/project path, you should be able to see the lists of projects created from Exercise 3 in JSON format. Otherwise, you can call the endpoint to see the project list.

Additionally, try to access the */auth* service from Postman. Send a POST request to localhost:8080/auth. In the Authorization tab, select Basic Auth type and input username and password (in plaintext). Send the request and see if the response body contains the expected value as programmed in Mission 4.

Figure 2 shows the same request, but this time with Basic Authentication in Postman. Note that there is a small difference between the responses in the browser and in Postman, which is the missing `details.sessionId` property in Postman. The reason is that HTTP Basic Authentication is a *stateless* method of authentication. When using the form login the server keeps track of the authentication state through a user session and the client sends the `sessionId` (in contrast to the credentials with Basic Authentication) on each request to identify himself. By default the embedded

[9] Request Header: https://www.baeldung.com/spring-rest-http-headers#1-individually
[10] Authentication Manager: https://spring.io/guides/topicals/spring-security-architecture

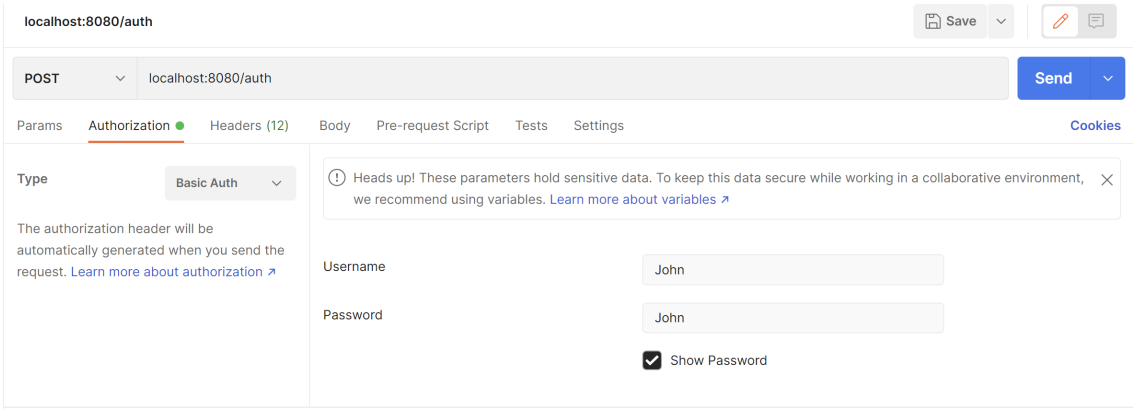

Figure 2: Basic Authentication HTTP Header in Postman

Tomcat stores HTTP session objects in memory. Typically, HTTP cookies are used to receive and send the `sessionId` as the browser handles storing and sending cookies by himself[11]. Spring uses a session cookie with the name `JSESSIONID` which you can see when looking at the cookie storage of the website in your browser's developer tools[12].

Mission 5: Apply Basic Authentication only for the `/auth` Endpoint

Basic Authentication should be applied during the authentication, however, if you use Postman to create a POST request to the localhost:8080/project, give the username and password using the Basic Authentication type, and send it to the server, you will get a list of projects. We do not want this, so let us restrict the Basic Authentication to only the `/auth` endpoint.

The key to solve our solution is Multiple HttpSecurity[13]. Basically, you can define two security configurations, one for the `/auth` endpoints and the other is for all other endpoints.

You can follow the technique mentioned in the footnote, or use another approach, e.g., creating two separate configuration classes. In the end, only the `/auth` endpoint can process the Basic Authentication request, as shown in Figure 4. Other endpoints (e.g. `/project`), despite having a valid user credential in Basic Authentication, shall return the login form and not let the user access the protected resources, as shown in Figure 3.

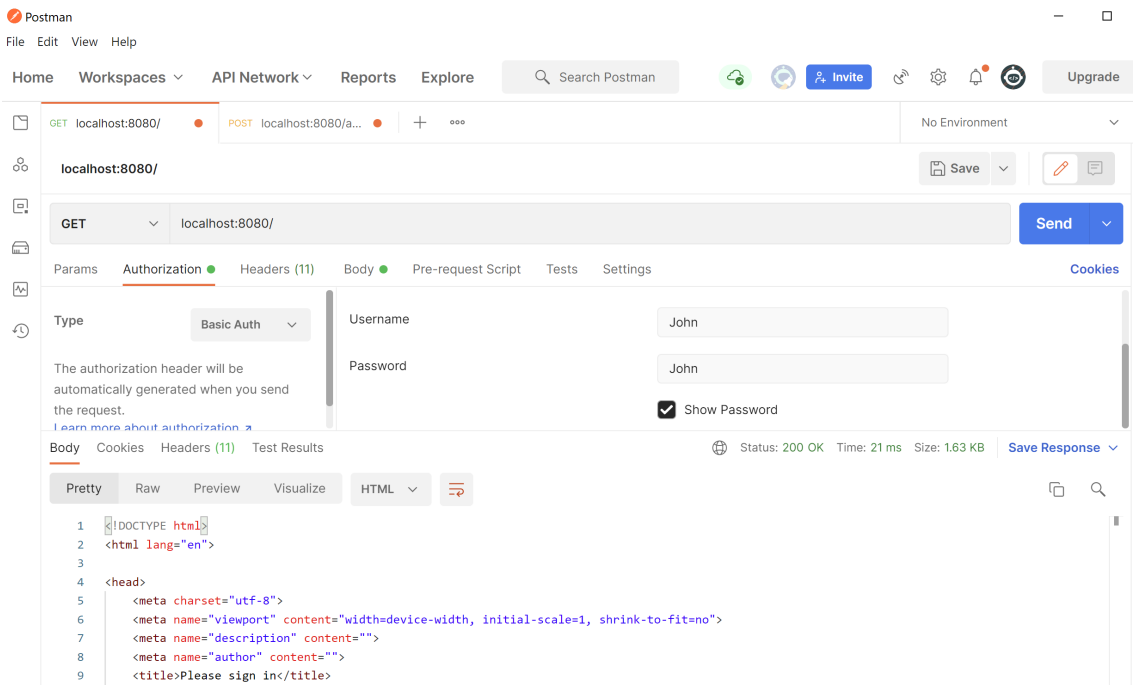

Figure 3: Access to endpoints other than `/auth` should return the login html. The output is a greeting as we previously programmed.

[11] HTTP Cookies: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
[12] HTTP Cookies in Chrome Developer Tools: https://developers.google.com/web/tools/chrome-devtools/storage/cookies
[13] Multiple HttpSecurity: https://docs.spring.io/spring-security/site/docs/current/reference/html5/#multiple-httpsecurity

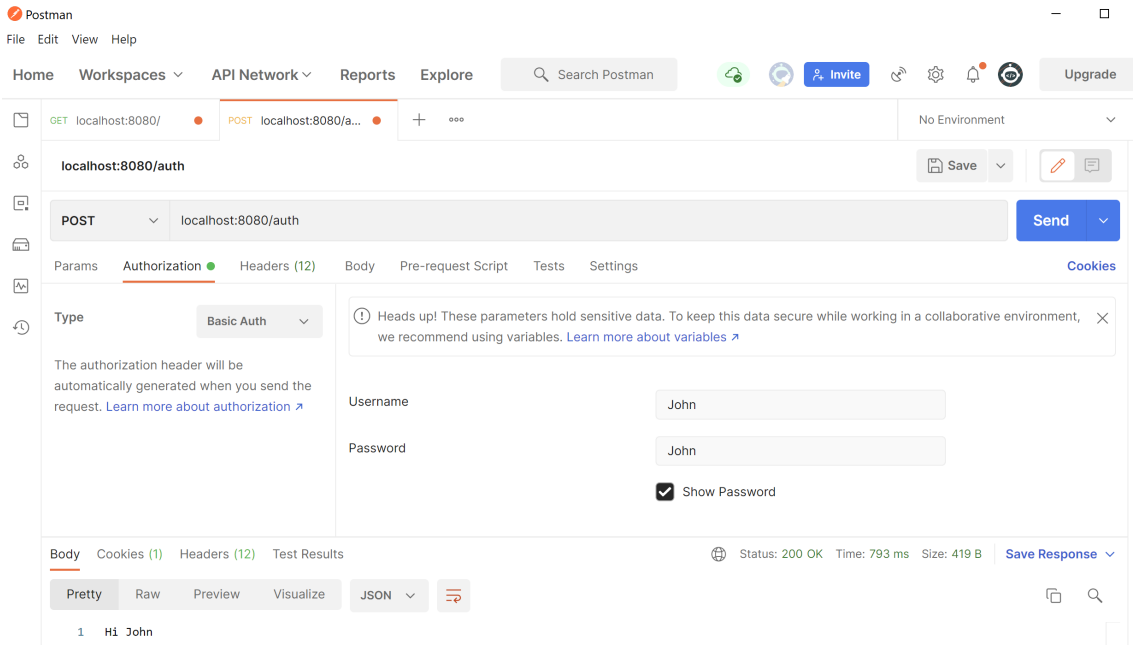

Figure 4: Basic Authentication is accepted at endpoint `/auth`. The output is a greeting as we previously programmed.

Exercise 3: Apply Authorization to REST Services

You have seen how we apply authentication to the project endpoints from the previous exercises. Yet, any user in our system can call any service without restrictions, so we should define an authorization policy for different user roles. For example, a project manager can create a new project, but a developer cannot.

With these assigned authorities (Project Manager and Developer), we can make use of the authorization features of Spring Security. In this exercise, we will see two ways to enable the authorization policies:

1. Using **@PreAuthorize** and **@PostAuthorize**[14]
2. Specify the permitted access role(s) for each endpoint pattern in the Spring Security Configuration.

To use the **@PreAuthorize** and **@PostAuthorize**, we simply add a `@EnableGlobalMethodSecurity` class annotation on a dedicated `@Configuration` bean. We can add this annotation in our `SecurityConfig` class:

```
@Configuration
// Other configs
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    // ...
}
```

Now, we can use them on methods of controllers, services, repositories, or other classes. The possibilities of using expression-based access control in Spring Security are exhaustive and we only show a simple example which requires the current `Principal` to have a specified authority. Let us define that only project manager can create projects in the system. This can be done as follow in the `ProjectController`:

```
@RestController
@RequestMapping("/project")
public class ProjectController {

    @Autowired
    ProjectService projectService;

    @PostMapping("")
    @PreAuthorize("hasAuthority('" + UserRole.MANAGER + "')")
```

[14] @PreAuthorize and @PostAuthorize: https://docs.spring.io/spring-security/site/docs/5.2.x/reference/html/authorization.html#access-control-using-preauthorize-and-postauthorize

```
    public Project createProject(@RequestBody Project project) {
        return projectService.createProject(project);
    }

    // Other project services
}
```

If you login as a user with the role `DEV`, executing POST `/project` will give you an HTTP 403: Forbidden error. For more details, please have a look at the built-in expressions in the Spring Security documentation at https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#el-access. Specifically, they allow using the `Authentication` object directly when defining access rules.

The other way to define the access/deny rules for REST services is by specifying the access role for each endpoint in the Spring Security config. First, let us comment out the `@PreAuthorize` and @EnableGlobalMethodSecurity(prePostEnabled = true), then navigate to the `SecurityConfig` class and enforce that a POST request to `/project` shall be permitted only for Project Managers.

```
@Configuration
// Config Annotators
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
            .authorizeRequests()
                .antMatchers(HttpMethod.POST,"/project").hasAuthority(UserRole.MANAGER)
            // other configurations
    }

    // ...
}
```

The behavior should be the same as using `@PreAuthorize` previously. **Note: the order of endpoints configured inside the `authorizeRequest` method affects the priority of the authorization rules**. Therefore, authorization policies for subpaths should be defined before the policies of the superpaths. For example, defining the authorization policy of POST `/project` after `/project/**` like below will grant access to the POST `/project` service for all authenticated users:

```
@Configuration
// Config Annotators
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
            .authorizeRequests()
                .antMatchers("/project/**").authenticated()
                .antMatchers(HttpMethod.POST,"/project").hasAuthority(UserRole.MANAGER)
            // other configurations
    }

    // ...
}
```

However, if the authorization configuration of POST `/project` is placed before the `/project/**` path, then Spring enforces the authorization policy correctly, i.e. developers cannot create projects in the system.

Exercise 4: Authenticate and Authorize Request with JSON Web Token (JWT)

In the previous exercise, you already got to know different approaches for authenticating and authorizing a user. Let's transfer this knowledge to something meaningful and secure an application's REST API which allows creating, listing and querying specific project entities. This will be necessary for the securing the back-end microservices of your ASEDelivery system.

Next, you will learn how to create a JSON Web Token (JWT)[15] to verify the user credential of a client request.

[15] JWT: https://jwt.io/introduction

JWT is a standard to define compact data transmission format between parties. JWT data is stored as a JSON object and signed using a secret or a public/private key pair. JWT has two types: Java Web Signature (JWS) and Java Web Encryption (JWE). For *JWS*, the sender signs the JWT token, and the receiver verifies the integrity with a key. In *JWE*, the data are encrypted to hide information from other parties.

In this exercise, we will apply JWS tokens to secure the services with a public/private key pairs (From now on, we will refer JWS token as JWT). The private key is used to generate a JWT token, while the public is used to verify whether a JWT is signed by the intended issuer. In this exercise, the issuer is the `Project` application. Our JWT has two main functions: authenticate the users by returning a JWT token after a successful authentication at `auth`, and authorize them whenever the users access project management services in the backend.

If you want to apply JWE, you can follow this tutorial https://medium.com/aeturnuminc/encrypt-and-decrypt-sensitive-data-with-jwe-70421722f7e5.

First, let us create a *KeyStore*[16] to hold the public/private key pair using *keytool* from Java. Basically, a Keystore manages three types of cryptographic keys: Private Key, Secret Key and Trusted Certificate (Public Key). Each key has an alias attached to them. The *Private Key* can be optionally protected by a password.

Open the Terminal (PowerShell/cmd in Windows) in the `src/main/resources` folder of the project and call the following command:

```
keytool -genkey -v -keystore ase_project.keystore -alias aseprojectkey -keyalg RSA -keysize 2048 -
    validity 10000
```

In this command:

- `-keystore` contains the location of the keystore file.
- `-alias` is used to refer to the public and private keys in the keystore.
- `-alg` is the encryption algorithm
- `-keysize` denotes the bit in the keys, 2048 is the recommended minimum value. Larger key size is less susceptible to brute-force attack, but the computation will take longer.
- `-validity` specifies the expiration in days for the key. Here our key will expire after 10000 days.

You will then be prompted to enter the password of the KeyStore, which is used to access the public key. Afterward, *keytool* will ask about the organization and location. You can skip these steps by pressing enter. **Note: In the end, you will be asked to enter the password for your private key**. If you want to use the same password as the public key, press enter. To maintain the simplicity of this exercise, we use the same password **aseproject** for both the private and public keys.

Next, we will create a package for our JWT Util at `src/main/java/edu/ase/project/jwt`, where we store the logic of generating and verifying JWT. Additionally, we also include a KeyStore manager to execute the loading and retrieval of the public/private key pair. The methods inside a KeyStore manager is only accessed by the JWT Util class.

```
@Component
public class KeyStoreManager {

    private KeyStore keyStore;

    private String keyAlias;

    private char[] password = "aseproject".toCharArray();

    public KeyStoreManager() throws KeyStoreException, IOException {
        loadKeyStore();
    }

    public void loadKeyStore() throws KeyStoreException, IOException {

        keyStore = KeyStore.getInstance(KeyStore.getDefaultType());

        FileInputStream fis = null;
```

[16] Java KeyStore: https://docs.oracle.com/javase/7/docs/api/java/security/KeyStore.html

```
        try {
            // Get the path to the keystore file in the resources folder
            File keystoreFile = ResourceUtils.getFile("classpath:ase_project.keystore");

            fis = new FileInputStream(keystoreFile);

            keyStore.load(fis, password);

            keyAlias = keyStore.aliases().nextElement();

        } catch (Exception e) {
            System.err.println("Error when loading KeyStore");
            e.printStackTrace();
        } finally {
            if (fis != null) {
                fis.close();
            }
        }
    }

    protected PublicKey getPublicKey() {
        try {
            // TODO: return the public key in the keystore
            return null;
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }

    protected Key getPrivateKey() {
        try {
            // TODO: return the private key in the keystore.
            return null;
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }
```

An key concept in JWT is *JWT Claims*, which store the data about the entity of interest, i.e., the user in our example. Additionally, *JWT Claims* can contain extra information that is relevant for verifying the JWT or executing services. There are three types of claims: Registered Claims, Public Claims, and Private Claims. Registered Claims are non-compulsory but recommended information such as the subject of the JWT (sub), expiry time (exp), and other tags. Public claims contain common data (gender, picture, etc.) with collision-resistant names. Private Claims stores custom data shared between parties, which you can leverage to store business-specific information. **If you are using JWS, do not include secret information as the JWT is readable to everyone**. JWE is suitable if you want to secure secret information from the public.

Before developing the JWT Utilities, let us import the necessary dependencies from **jsonwebtoken**, a library to process JWT, into our *pom.xml* file:

```
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-api</artifactId>
            <version>0.11.2</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-impl</artifactId>
            <version>0.11.2</version>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-gson</artifactId>
            <version>0.11.2</version>
            <scope>compile</scope> <!-- Not runtime -->
        </dependency>

        <dependency>
            <groupId>io.jsonwebtoken</groupId>
```

```
            <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
            <version>0.11.2</version>
            <scope>runtime</scope>
        </dependency>
```

Next, we will create a JWT Utilities to generate and verify JWT tokens in the same package `src/main/java-/edu/ase/project/jwt`.

```
@Component
public class JwtUtil {

    @Autowired
    private KeyStoreManager keyStoreManager;

    public String generateToken(AseUser userDetails) {
        Map<String, Object> claims = new HashMap<>();
        claims.put("roles", userDetails.getAuthorities());
        return createToken(claims, userDetails.getUsername());
    }

    // Create JWS with both custom and registered claims, signed by
    // a private key.
    private String createToken(Map<String, Object> claims, String subject) {

        String jwt = Jwts.builder()
            .setClaims(claims)
            .setSubject(subject)
            .setIssuer("aseproject")
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 5)) // Expires
                after 5 hours
            .signWith(keyStoreManager.getPrivateKey(), SignatureAlgorithm.RS256)
            .compact();

        return jwt;
    }

    // Create a Parser to read info inside a JWT. This parser use the public key
    // to verify the signature of incoming JWT tokens
    private JwtParser loadJwtParser() {
        return Jwts.parserBuilder()
                .setSigningKey(keyStoreManager.getPublicKey())
                .build();
    }

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    private Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token)  {
        return loadJwtParser()
            .parseClaimsJws(token)
            .getBody();
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    // Check if the JWT is signed by us, and is not expired
    public boolean verifyJwtSignature(String token) {
        // TODO: Check if the JWT is signed by the public key in the KeyStore
        // and the JWT is not expired
        return null;
    }
}
```

The `generateToken()` method inserts a custom Claim (roles), whereas the `createToken()` method

creates the JWTToken with Registered Claims (subject, issuer, expiration, and issuedAt).

To read JWTs, we construct a JwtParser and attach the public key in our KeyStore as the signing key. The signing key helps us verify whether a JWS is signed by a trusted, intended party, i.e., our Project Management application.

The next four methods reveal how to extract all or a single claim inside a JWS token. Furthermore, we have a function to verify if a JWT is expired. Your mission is to complete the verification of the JWT Signature by checking if the JWS is not expired and signed by the public key in the KeyStore.

Now is the time to apply the JWT generation and verification methods. In our authentication service, we will generate and return a JWT in the response when the user successfully logs in:

```
@RestController
public class AuthService {

    // ...

    @Autowired
    private JwtUtil jwtUtil;

    public ResponseEntity<String> authenticateUser(
            String authorization,
            HttpServletRequest request) throws Exception {

        // ...

        // TODO: find if there is any user exists in the database based on the credential.
        // Hint: What do you get from Mission 3?
        final UserDetails userDetails = null;

        if (bcryptPasswordEncoder.matches(password,
                userDetails.getPassword())) {

            // Remove the setAuthentication() as we do not want to keep the
            // user authenticated in the upcoming requests. The user has to attach
            // our generated JWT for every request they sent to the server.
            // setAuthentication(userDetails, request);

            // TODO: Generate a JWT token based on the info of the authenticated user
            final String jwt = null;

            return new ResponseEntity<String>(jwt, HttpStatus.OK);

        } else {
            return new ResponseEntity<String>(
                    "Email or password is incorrect",
                    HttpStatus.BAD_REQUEST
            );
        }
    }
}
```

At this point, we do not have any mechanism to read the JWT token from a request. Therefore, let us create a custom `Filter` to extract the JWT from incoming requests, and authenticate the user if their JWT is valid.

In `src/main/java/edu/ase/project/filter`, create an `AuthRequestFilter` that extends `OncePerRequestFilter` which let us customize an execution on every request dispatch.

```
@Component
public class AuthRequestFilter extends OncePerRequestFilter {
    @Autowired
    private MongoUserDetailsService mongoUserDetailsService;

    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private AuthService authService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
        FilterChain filterChain)
            throws ServletException, IOException {
```

```
        String username = null;
        String jwt = null;

        final String authHeader = request.getHeader("Authorization");

        System.out.println("Authenticate Header " + authHeader);

        if (authHeader != null && authHeader.startsWith("Bearer")) {

            // TODO: Get the JWT in the header.
            // If the JWT expires or not signed by us, send an error to the client
            // TODO: Extract the username from the JWT token.

            jwt = null;

        } else {
            // No valid authentication, No go
            if (!authHeader.startsWith("Basic")) {
                response.sendError(HttpStatus.BAD_REQUEST.value(), "No JWT Token or Basic Auth
                    Info Found");
            }
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

            // TODO: load a user from the database that has the same username
            // as in the JWT token.
            User userDetails = null;

            authService.setAuthentication(userDetails, request);

            Authentication authContext = SecurityContextHolder.getContext().getAuthentication();

            System.out.println(String.format("Authenticate Token Set:\n"
                    + "Username: %s\n"
                    + "Password: %s\n"
                    + "Authority: %s\n",
                    authContext.getPrincipal(),
                    authContext.getCredentials(),
                    authContext.getAuthorities().toString()));

        }
        filterChain.doFilter(request, response);
    }
}
```

Finally, let us test our JWT authentication. First, send a Basic Authentication to the `/auth` service. If your implementation is correct, you will see a JWT in the response body. In any follow-up requests to the project services, attach the JWT into an **Authorization** header with the value as `Bearer <your-JWT>`. For example, send a GET request to `/projects` for viewing all existing projects with a new request header. Add a **Authorization** header into the GET request in Postman as Figure 5.

If you can see the list of projects, congratulation! In the next exercise, we will store JWTs not in the response body or local storage, but in a HttpOnly Cookie, so that malicious code cannot read it with Javascript like it does to local storage. Additionally, we will also implement CSRF and CORS protection policies.

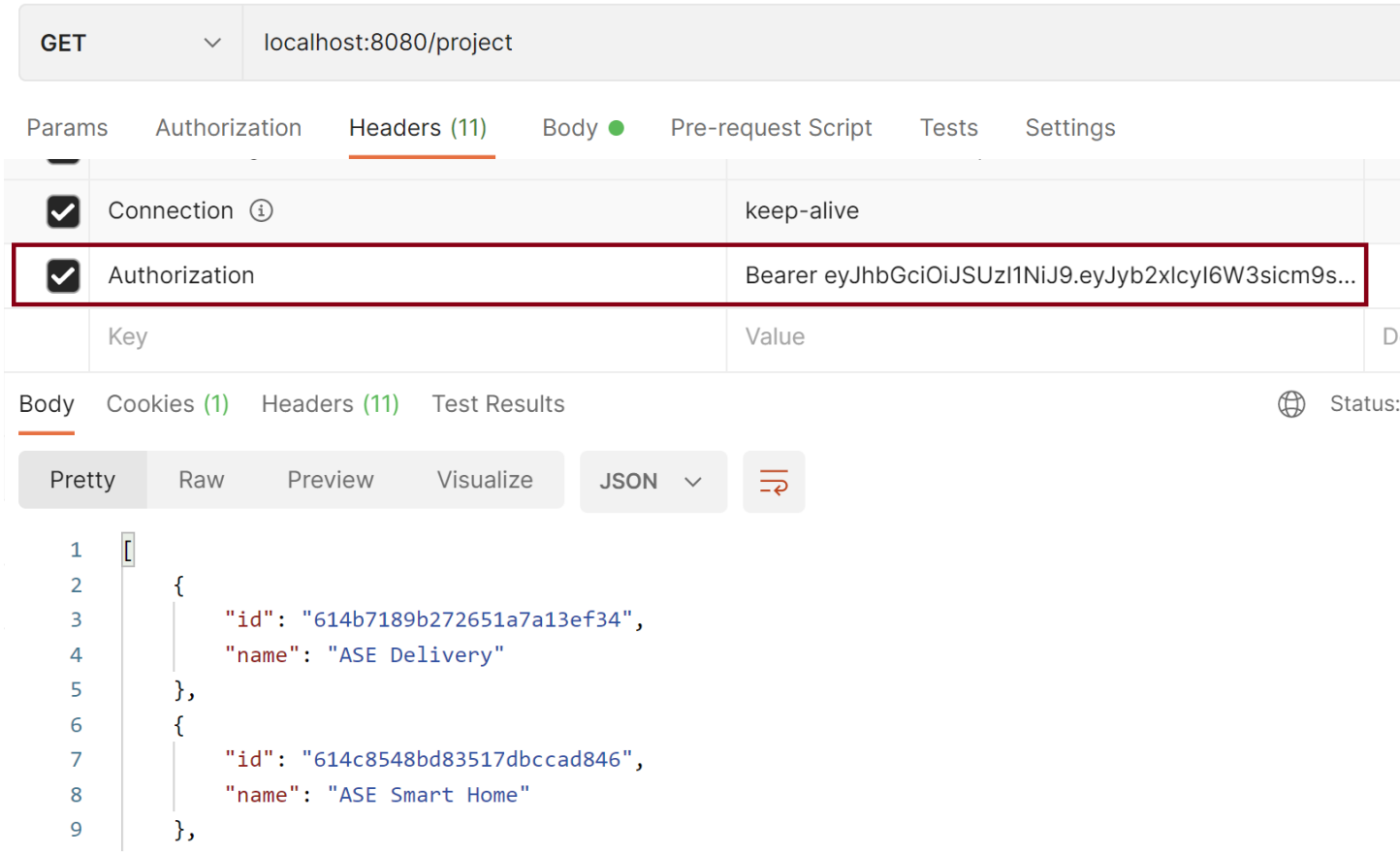

Figure 5: The list of projects are returned when a valid JWT Token is included in the *Authorization* header of the request