Spring Security
Lab 6

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

2 Overview of security options for REST APIs in Spring Security

There are several broad approaches to accomplishing this.

For authentication of users:

a) We can provide a facility for users to register an account for them to authenticate themselves (typically via username - password authentication). Next, we can explicitly configure the authentication mechanism for them as well as assign authorization rights for them to access specific REST resources in the main security configuration file (in the WebSecurityConfigurerAdapter class)

b) We can authenticate users via 3rd party OIDC / OAuth2 providers, obtain the relevant personal information from these providers and then either use the default authorizations assigned to them or grant them our custom authorizations: OAuth2 / OIDC

c) Authenticate using HTTP Basic Auth

For cases a) and b) above, users will initially authenticate via a custom login form.

Alternatively, users could register an account first and then authenticate via Basic Auth / Digest Auth instead of going through the custom login form. Then all subsequent requests will also be authenticated the same way.

For the case of a) and b) previously, after an initial authentication, clients will be provided with either an API key or a JWT token which they will include in all their subsequent requests to the service. This key or token will be used to authenticate these requests and also identify the specific authorization constraints that apply to the authenticated user.

For the case of an API key, the API key is either a CSPRNG nonce or a UUID which is linked to the user info stored in the database table. Any subsequent requests that include the API key will then involve extracting the API key and then looking up the matching user identity and associated authorizations in the table.

For the case of a JWT token, the required identity and authorization info can be already included in the token itself without the need for a subsequent table lookup. In that case, there are two possible general approaches to generating and using a JWT token to secure a REST API:

A more comprehensive approach would involve the following general steps (not covered in these labs)

a) Configure a custom authorization server and resource server (or utilize one provided by a 3rd party provider)
b) Clients then register with the authorization server and subsequently obtain an access token (and a refresh token optionally) through the standard OAuth2 workflow
c) Client present token to retrieve a protected REST API resource using an access token.

A simpler approach would be to have the REST service itself generate a simplified JWT token (without the additional overhead of an authorization server / resource server) and then return the token to the client to use and submit with all subsequent requests after the initial authentication

3 Implementing HTTP Basic Auth

The main folder for this lab is `BasicAuth-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `BasicAuthDemo`
Group: `com.workshop.security`
Artifact: `BasicAuthDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo working with Http Basic Auth`
Package: `com.workshop.security`

Add the following dependencies:
`Web -> Spring Web`
`Security -> Spring Security`

```
Developer Tools -> Lombok
Developer Tools -> Spring Boot DevTools
```

In the package `com.workshop.security` in `src/main/java`, place these files:

```
MainController
Developer
SecurityConfig
```

The primary configuration to enable HTTP Basic Auth is in the main security configuration class. The key points

- There are 4 preconfigured username / password combinations for in-memory authentication. For a real-life application, we can read these combinations from a persisted database table as covered in a previous lab.
- We disable CSRF protection (i.e. removing the need to submit the CSRF token with state-modifying requests such as POST, PUT, etc). This is because the mechanics of constructing a RestTemplate client programmatically makes it invulnerable against the browser cookie-based attacks that CSRF is based on.
- We disable session management (i.e. there is no session cookie used to track client interaction with the browser). This is because REST API architecture by its nature should be stateless.
- We enable HTTP Basic Authentication

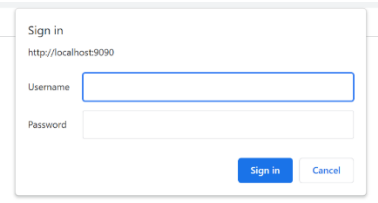
The MainController class implements 3 standard @GetMapping methods which are mapped to 3 different paths.

Start up the Eclipse TCP/IP monitor and run it on port 9090 (or any other free port) and redirect the traffic to port 8080.

Start the app in the usual manner from the dashboard.

In a browser tab, navigate to:
http://localhost:9090/api/first

Notice that the initial response from the server returns a status code of 401 with the `WWW-Authenticate` header set to `Basic realm="Realm"`. This is the standard response when no username / password combination is submitted in the initial request, and informs the browser to produce a pop-up dialog box to acquire the required credentials.



Type in any one of the valid username / password combinations and click Sign in.

Notice now that this combination is sent in Base64 form in the Authorization header in the GET request back to the service, for e.g.

```
Authorization: Basic c3BpZGVybWFuOnNwaWRlcg==
```

Use any online Base64 decoder to verify this
https://www.base64decode.org/

The response from the service now includes the JSON serialization of the Developer object returned from the handler method mapped to /first in MainController. Notice as well that there is no session cookie returned, because session tracking has been disabled.

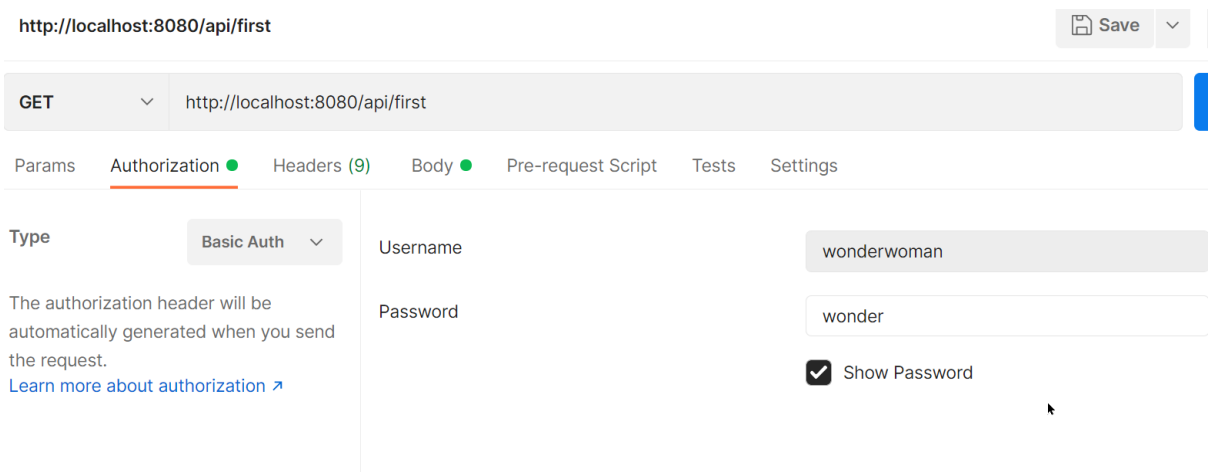
If you refresh the tab again, you will see the same request sent again with the same username / password combination in the Authorization header. This is because the browser automatically caches this username / password combination.

Clear the browsing data in the browser and refresh the tab again. Since the cache is now cleared, no username / password combination is now submitted with the request, whereby the response from the server again returns a status code of 401 with the `WWW-Authenticate` header set to `Basic realm="Realm"`, resulting in the production of a pop-up dialog box to acquire the required credentials again.

Using Postman, make a request to:
http://localhost:9090/api/first

Notice the standard JSON error message is returned with a 401 status code as well as the `WWW-Authenticate` header set to `Basic realm="Realm"`,

Now make a request with the Authorization send to Basic Auth, and with an appropriate username / password combination:



This time a valid JSON response is returned.

At this point of time, we have not enforced any specific authorization constraints related to roles on the 3 paths (`/api/first`, `/api/second` and `/api/third`) available in the MainController.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`SecurityConfig-v2`

Restart the app.

Make requests to the service for the 3 different paths using different username / password combinations and verify that the appropriate status code 403 messages are returned as a result of requests to resource paths that the specific username / password combination are not authorized to access.

We can also further customize the default error messages that are returned when

- A request is made without the appropriate Authorization header containing the necessary username / password
- A request is made for a resource that is not authorized for the authenticated user

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`SecurityConfig-v3`

In the package `com.workshop.security` in `src/main/java`, add the following files:

```
CustomAccessDeniedHandler
CustomAuthenticationEntryPoint
CustomErrorMessage
```

These are the two key issues of HTTP Basic Auth for securing REST APIs:

- Since the username / password is transmitted in Base64 form (which is easily decoded), the entire connection must be secured using HTTPS
- The username / password is automatically cached by the browser in order to be resent with all subsequent requests after the initial successful authentication. This gives significant risk to the compromise of the password by a malicious script that might be running in the browser from another domain. This situation is even more problematic than when cookies are used to track sessions since the cookie does not actually contain the original username / password combination. Therefore, theft of a session cookie will only allow access to the user's resources for the duration of a single session that the cookie is valid for, while theft of the username / password results in permanent compromise of the user's resources.

4 Implementing JWT for securing a REST API

The main folder for this lab is `JWT-REST-Demo`
Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `JWTRESTDemo`
Group: `com.workshop.security`
Artifact: `JWTRESTDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo working with JWT to secure a REST API`
Package: `com.workshop.security`

Add the following dependencies:
`Web -> Spring Web`

```
Security -> Spring Security
Developer Tools -> Lombok
Developer Tools -> Spring Boot DevTools
```

Add in these additional dependencies in the project POM to allow us to use JJWT. Do a Maven -> Update Project.

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

In the package `com.workshop.security` in `src/main/java`, place these files:

```
ControllerExceptionHandler
CustomErrorMessage
JwtAuthenticationController
JwtRequest
JwtResponse
JwtTokenUtil
MainSecurityConfig
```

`JwtAuthenticationController` provides the main logic to extract the username and password from an incoming request in the format of JwtRequest to the path `/authenticate`. It uses JwtTokenUtil to generate the JWT which contains the following claims: (`iat`, `sub`, `iss`, `role`). This is returned to the client via `JwtResponse`

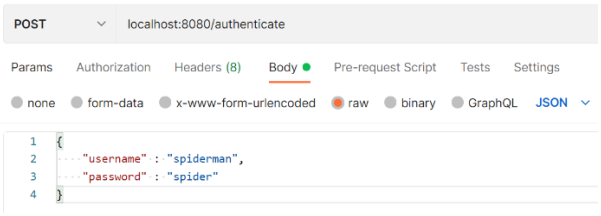
`ControllerExceptionHandler and CustomErrorMessage` both provide custom JSON error response to an authentication error

`MainSecurityConfig` is configured to permit all requests through to the `/authenticate` endpoint, to allow custom authentication, and also disables CSRF and session tracking for a REST API.

Start the app from the Boot dashboard

Using Postman, send a password / username combination with the following POST request:

```
POST localhost:8080/authenticate
```



Verify that the token received in the response has the appropriate subject and role claims, based on the hardcoded configuration in `MainSecurityConfig.` You can use any online JWT decoder: https://jwt.io/

Verify that the customized error message is returned for invalid username / password combinations. Notice that we are using an @ExceptionHandler in a @ControllerAdvice class to handle the exception and return a CustomErrorMessage because the original exception is not a specific Spring-security related exception (AuthenticationException or AccessDeniedException), so we can handle it like a typical REST related exception

We now need to extend this app to add in a filter that performs authentication / authorization of a future incoming request that includes this token in the Authorization: Bearer header.

In the package `com.workshop.security` in `src/main/java`, place these files:

```
CustomAccessDeniedHandler
JwtRequestFilter
MainController
Developer
```

In the package `com.workshop.security` in `src/main/java`, make these changes:

```
MainSecurityConfig-v2
```

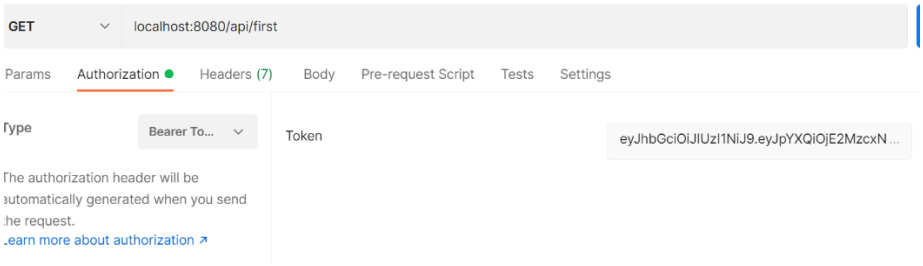
In JwtRequestFilter, we extract the token from the Authorization header, validate and parse it, and then create a new UsernamePasswordAuthenticationToken using the subject name and role retrieved from the token. This is then used to set the SecurityContext before the request gets passed on to the next filter in the filter chain (as configured in MainSecurityConfig).

Start the app from the Boot dashboard

Send a POST request as before with any suitable valid username / password combination:

```
POST localhost:8080/authenticate
```

Create a new GET request to any one of the following valid API endpoints with the Authorization set to Bearer type and paste in the JWT token obtained from the previous POST request:



Verify that you are able to obtain back the specific results from the various handler methods in the MainController class in accordance to the specific roles assigned to the token from the initial authentication in the first POST request. Notice now that we are also returning a custom error message when a request is made for a resource that is not authorized for the submitted token.