Securing APIs

Module 2 - Week 8

Schedule

Week	**Topics**
Week 0	Module 2 Orientation / PostgreSQL
Week 1	Intro to databases / Ordering, limiting, and grouping
Week 2	SQL joins / Insert, update, and delete / Database design
Week 3	Data Access / Data security
Week 4	DAO testing
Week 5	**Mid-module project**
Week 6	Postman / NPM / Networking and HTTP / Consuming RESTful APIs
Week 7	Server-side APIs
Week 8	Securing APIs
Week 9	**End-of-module project**
Week 10	**Assessment**

Objectives

- **Define and differentiate between the terms "authentication" and "authorization" as they pertain to a client-server or Web application**
- Describe the general mechanics and workflow of how to use JSON Web Tokens (JWTs) to authenticate users of a client-server (including Web) application
- Utilize the auth features of an application framework (Spring Boot)
    - Specify a particular resource with anonymous access, that doesn't require authentication
    - Specify a particular resource that requires authentication for access
    - Apply simple authorization rules for resources
    - Obtain the identity of an authenticated user

Authentication/Authorization

Authentication is the act of validating that users are whom they claim to be. This is the first step in any security process.

Common forms of authentication:

- Username/Password
- One-time pins
- Authentication apps.
- Biometrics

Authorization is the process of giving the user permission to access a specific resource or function.

HTTP

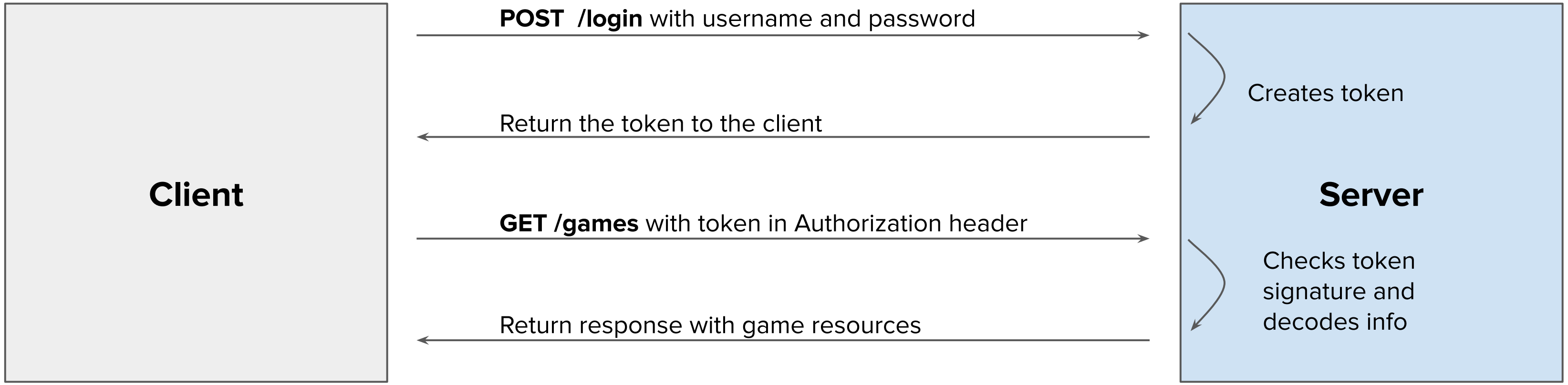
The HTTP Status Codes associate with authentication and authorization fall into the 4xx (client-related) group of statuses.

401 Unauthorized indicates that the user has not been authenticated

403 Forbidden indicates that the user is authenticated, but is not authorized to access the resource specified in the HTTP request.

Authentication Workflow

- On login the server generates a secure token with a JSON payload and a signature.
- The token payload contains information about the user, when it expires, and what the user is allowed to do (authorizations).
- The token is returned to the client.
- The client then sends this token with each further request, the server can verify it was created by the server using the signature.
- Token goes in header with word `Bearer`, which indicates type of authentication



POST /login with username and password
Creates token
Return the token to the client
Client
Server
GET /games with token in Authorization header
Checks token signature and decodes info
Return response with game resources

JWT

- JSON Web Tokens (JWT) are an Internet Standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- Allows for a client to identify itself to the server as having already been authenticated allowing for multiple requests after login.
- This allows for multiple requests to occur after a single login regardless of the stateless nature of HTTP and REST.
- Compact size allows for quick transfer with requests.
- Often used as authorization mechanism, storing user info such as permissions or roles in payload. These are called claims.
- Can contain any data that can be represented in JSON.
- JWT actually contains JSON, but it's encoded.

`https://jwt.io/`

HttpEntity

`HttpEntity` allows us to package information for an **HTTP** request such as the headers or body.

```
private HttpEntity<Void> makeAuthEntity() {
    HttpHeaders headers = new HttpHeaders();
    headers.setBearerAuth(authToken);
    HttpEntity<Void> entity = new HttpEntity<>(headers);
    return entity;
}
```

```
Hotel[] hotels = restTemplate.exchange(API_BASE_URL + "hotels",
     HttpMethod.GET, makeAuthEntity(), Hotel[].class).getBody();
```

Using **`exchange`** here allows us to attach an **`HttpEntity`** with **`AUTH_TOKEN`** in **`Bearer Auth`** header even though **`GET`** has no body to attach headers to in the **`HttpEntity`**

ResponseEntity

`ResponseEntity` allows us obtain a package of information from an **HTTP** response such as the headers or body.

`exchange` returns a **`ResponseEntity`** of a specified type rather than an object of the actual type.

```
try {
    ResponseEntity<TokenDto> response = restTemplate.exchange(API_BASE_URL + "login",
        HttpMethod.POST, entity, TokenDto.class);

    TokenDto body = response.getBody();
    if (body != null) {
         token = body.getToken();
    }
} catch (RestClientResponseException | ResourceAccessException e) {
    BasicLogger.log(e.getMessage());
}
```

Authorization

- Authorization is the process of giving a user permission to access a specific resource or function.
- It's not uncommon to have certain resources or functions only be accessible to certain people or roles.
- We can authorize access by the user's role (i.e. only ADMIN users can access the resource)
- We can define rules for each resource. These rules can:
  - Allow anonymous access
  - Require authentication
  - Grant access based on user role

Authorization Rules

- Authorization rules can be specified for the entire class by adding the `@PreAuthorize` annotation on the class itself.

- We can override rules at the method level:
    - `@PreAuthorize("isAuthenticated()")`
      The user must be authenticated.
    - `@PreAuthorize("permitAll")`
      The user doesn't have to be authenticated. Anonymous (non-authenticated) access.
    - `@PreAuthorize("hasRole('ADMIN')")`
      The user must be authenticated and have the role ADMIN.
    - `@PreAuthorize("hasAnyRole('ADMIN', 'USER')")`
      The user must be authenticated and have either the ADMIN or USER role.

Principal

- There are times where you'll need access to the current logged in user.
- For example, you have secured the `delete()` method to users with the role ADMIN, but what if you wanted to keep an audit log of which user deleted each reservation?
- Spring gives you access to information about the current user if you add an argument of type `Principal` to your method.
- When you annotate a method with `@RequestMapping`, that method has a flexible signature, and you can choose from a range of supported controller method arguments.

```
@RequestMapping(path = "/currentuser")
public String currentUser(Principal principal) {
    String userName = principal.getName();
    log("Current user is " + userName);
    return userName;
}
```