Sam Hersick, Aidan Marshall
Dr. Hoang Bui
CS*466*01
8 December 2025

Backend Authentication and Authorization Using Spring Security

Our final project covers the topics of authentication and authorization. We chose this project because we wanted to better understand how backend servers, specifically REST APIs, are secured. For our tech stack, we selected Java 21, Maven, Tomcat, and MySQL. We selected Spring Boot and Spring Security for our backend framework based on industry relevance. For the coding portion, we decided to build a RESTful API for managing a basketball league. This allowed us to explore the topics of Spring Architecture, HTTP Authentication, Stateless Authentication, and Role-based Authorization through a realistic application. Our basketball management platform hosts 2 user roles: “user” and “admin”. Users have access to read endpoints, while administrators have additional access to create, update, and delete endpoints. Example: a user can view game results (teams, points, location, date), while administrators have the ability to schedule a future game or update the results of a completed game.

We quickly realized that working with a framework, especially as detailed as Spring Boot, had a steep learning curve. It took 40+ hours of watching YouTube videos, reading blog articles, and studying documentation to finally understand the existing functionality of Spring Boot and how to correctly customize it to meet our application requirements.

In terms of the Basketball Management platform itself, we were able to develop a normalized database Schema (seen in Schema.sql), for storing Teams, Games, and Players from the basketball league, as well as Users and their Authorities from the online platform.

We connected our REST API to our SQL database using JDBC. We configured this MySQL connection in application.properties following Spring Boot best practices. We then used Spring’s JdbcTemplate object to write custom SQL queries to our database. During the development of our repository layer, we learned our first backend security lesson: **Prepared Statements**. PreparedStatements are the industry-standard way to secure SQL interactions in Java. SQL Injections are a common attack against API’s. They work by injecting SQL code (ex: DROP TABLE users;) in their http requests. PreparedStatements separate queries and parameters, defending against these malicious API requests.

Once we finished developing a secure data access layer, we created several endpoints in our Controller layer. The controller layer is how our API maps incoming HTTP requests to backend logic. In Spring Boot, all controller classes are annotated with @RestController, telling Spring, “Hey, here are my endpoints”! By this point, we had created several endpoints that allowed us to begin the objective, “How can we secure our API through authentication and authorization?”

At Spring Security's foundation, we have the **SecurityFilterChain** that is responsible for filtering all API requests by performing auth before allowing it to reach our backend. SecurityFilterChains consist of a series of filters (either preexisting in Spring Security or custom built) that each perform a specific task. Some instances of these filters include:

- CSRF Filter - Protecting against CSRF attacks
- Basic Authentication Filter - Perform Basic HTTP Authentication
- JWT Authentication Filter - Authenticate user from signed JSON Web Token
- ExceptionTranslationFilter - Translates Java exceptions into meaningful HTTP Responses
- AuthorizationFilter - Authorize user action based off endpoint and user roles

A vital aspect of Spring Security Architecture is the **SecurityContextHolder**. This object contains user information that gets updated by each filter throughout the chain. Whenever it comes time to finally authenticate the user or authorize the specific request, Spring Security uses the information stored within the SecurityContextHolder to make these vital decisions.

Principal - Identifying Information (**username**)
Credentials - User's **password**
Authorities - their roles (**user**/**admin**)

Because the security context gets updated sequentially and overridden by downstream filters. **The order of filters in the filter chain is vital**. Example: You need to authenticate a user before you try to authorize them for a specific endpoint.

Let's finally dive into the specific security implementations that made our project special! First of all, we implemented a completely custom **JWT Authentication Service** that allowed our backend to be modern and stateless (sessions aren't recorded in the database). When a user registers (/register) or logs in (/login), on success, they are returned two JWT tokens: an **access token** and a **refresh token**. The access token expires after 10 minutes and is passed in all future HTTP requests to authenticate the user. The refresh token expires after 30 days, and whenever an access token expires, the client can send the refresh token to the /refresh endpoints to retrieve a new valid access token! This further secures our API and allows the user to avoid sending the username/password every time.

Next, we used BCrypt as our Password Encoder. This allows us to securely store passwords in our database. BCrypt uses Salt and a strong hashing algorithm to ensure confidentiality. Lastly, we included both **URL-based Authorization** as well as **Method-level Authorization**.

URL-based authorization for all “player” endpoints (loyola/basketball/Config/SecurityConfig.java)

```
http.authorizeHttpRequests((req) -> req
        .requestMatchers( ...patterns: "/team/**", "/game/**").authenticated()
        .requestMatchers( ...patterns: "/player/**").hasRole("ADMIN") // Only Admins can access players directly
        .anyRequest().permitAll()
);
```

Method-level authorization for the **DELETE** endpoint in team (loyola/basketball/Controller/TeamController.java)

```
@DeleteMapping()
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<?> delete(@RequestParam int ID){
    // Delete team from database
    // ...

    return ResponseEntity.status(HttpStatus.OK).body("Team "+ID+" Successfully Deleted");
}
```

Beyond the application we built, we gained a much deeper understanding of security vulnerabilities and how to defend against them. Spring Security stands out not only for its strong default protections but also for its flexibility when integrating modern authentication and authorization tools. This final section highlights key security concepts we learned that extend beyond the scope of our implemented features.

Spring Security has built-in protections against common attacks done by malicious users. The Cross Site Request Forgery (CSRF) attack tricks your browser by making a request to a legitimate website when you did not intend for this request to happen. A malicious program hides code in order to make this request. It is important to note that a CSRF attack is possible only when using stateful authentication. This is because the browser automatically attaches session cookies to every request. An example of a CSRF attack would be a program that makes a request to steal money from a user’s bank account.

This attack is possible when there is no way to distinguish whether it was the application or the legitimate user’s browser who made the request. Because of this, Spring Security uses a Synchronizer Token Pattern to validate that the request is coming from the correct session. When a session is open, a unique CSRF token is generated on the server. The incoming requests must include the correct CSRF token to validate that the request is coming from the intended website. CSRF tokens are stored on the server and manually inputted in forms when requests are made. Attackers cannot read or access the CSRF token from the server or the HTML from another site, removing the threat of CSRF attacks.

A Cross-Site Scripting (XSS) attack is when a user injects malicious JavaScript code, the script is intended to make it into the HTML of the website, and the web application fails to clean the input before it is run in the browser. XSS attacks can be delivered through the URL of a website, forms, or data from the database. XSS attacks are dangerous because the JavaScript that is injected can access variables stored in the client, and also can display anything on the browser. In order to prevent XSS attacks, a template engine like Thymeleaf applies automatic HTML escaping by default. This means that it converts characters that could run scripts (e.g. “<” or “>”) into safe text (“&lt” and “&rt”).

Although we decided to build a RESTful API application, we still explored these concepts that we were captivated by. These concepts were partially inspired by David Opitz's guest lecture. One of the topics that we decided to explore was OAuth2. A common misconception about OAuth2 is that it is a login system that allows you to login to a website. OAuth2's goal is to allow one program to access specific data from another application without having to share a password. The value that OAuth2 provides is security, simplicity, and speed. Users trust specific websites with their password, and they may not trust others. OAuth2 allows them to be certain that their password is protected. It is also significantly faster to use OAuth2 compared to having to create a new account for a separate website. Finally, no new passwords have to be remembered or stored when using OAuth2. Popular implementations of OAuth2 are "Login with Google" or "Login with iCloud".

If we had more time to work on the program for our project, our next goal would be to implement multi-factor authentication (MFA). From David Opitz's lecture, we learned that in order to have multi-factor authentication you must possess two out of three authentication factors. The three factors are something you know, something you have, and something you are. An example of something you know is a password. This is the most typical example in single-factor authentication, but passwords are not enough to protect your account. That is why something you have and something you are is important. An example of something you have is an authenticator app. Since only you have access to your authenticator app, no one else knows the code needed to login to your account. Finally, an example of something you are is Face ID. Biometrics are generally the hardest for hackers to bypass, but can be difficult for anyone besides large companies to implement.

References:

https://docs.spring.io/spring-security/reference/index.html
https://docs.spring.io/spring-security/site/docs/current/api/
Servlet Authentication Architecture :: Spring Security
https://www.youtube.com/watch?v=_GSHvvken2k
Spring Security Filter Chain Explained in 3 Minutes
https://www.youtube.com/watch?v=GH7L4D8Q_ak
https://www.jwt.io/introduction#what-is-json-web-token-structure
Managing JWT Refresh Tokens in Spring Security: A Complete Guide - Java Code Geeks
🔁 Refresh Tokens & JWT Expiry: The Complete Guide with Spring Boot and React | by Vishwanath Patil | Medium