# Spring Security

***<u>Role of Spring Security within the Spring Ecosystem</u>***

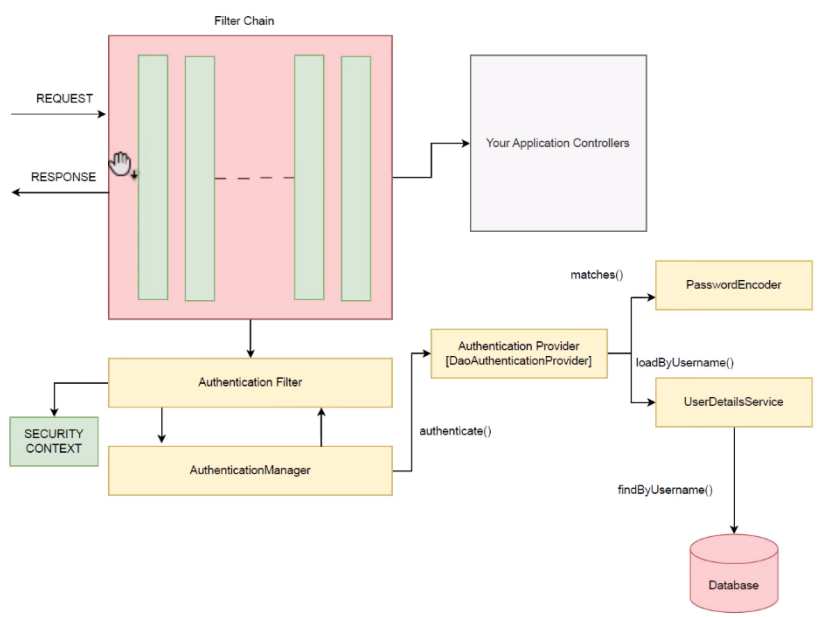
→ *Spring Framework*

→ *Spring Boot*

→ *Spring Data*

→ *Spring Security*
- *Authentication*
- *Authorization*

## How Spring security works?




---

### 1. Request and Filter Chain

- **Request**: The process starts when a user sends a request to the application.
- **Filter Chain**: The request passes through a series of **filters** in the Spring Security filter chain. Each filter performs specific security-related tasks (e.g., authentication, authorization).

---

## 2. Authentication Filter

- The **Authentication Filter** intercepts the request to handle authentication.
- It extracts the credentials (e.g., username and password) from the request and forwards them to the **AuthenticationManager** for verification.

---

## 3. Authentication Manager

- The **AuthenticationManager** delegates the authentication task to an **Authentication Provider**.
- This layer allows flexibility, as multiple authentication providers can be configured (e.g., for different user stores like databases, LDAP, etc.).

---

## 4. Authentication Provider (DaoAuthenticationProvider)

- The **DaoAuthenticationProvider** is a commonly used authentication provider in Spring Security.
- It interacts with two main components:
  - **PasswordEncoder**: Compares the raw password from the request with the encoded password stored in the database.
  - **UserDetailsService**: Retrieves user details (e.g., username, password, roles) by calling the `loadByUsername()` method.

---

## 5. UserDetailsService and Database Interaction

- **UserDetailsService**: Implements the `loadByUsername()` method to fetch user details from the database.
- The database is queried (via `findByUsername()`) to retrieve the user's data, such as username, password, and roles.

---

## 6. Password Matching

- The **PasswordEncoder** ensures the password provided by the user matches the encoded password stored in the database.

---

## 7. Security Context

- If the authentication is successful:
  - The authenticated user details are stored in the **Security Context**.
  - This context is used throughout the application to manage the user session and access control.

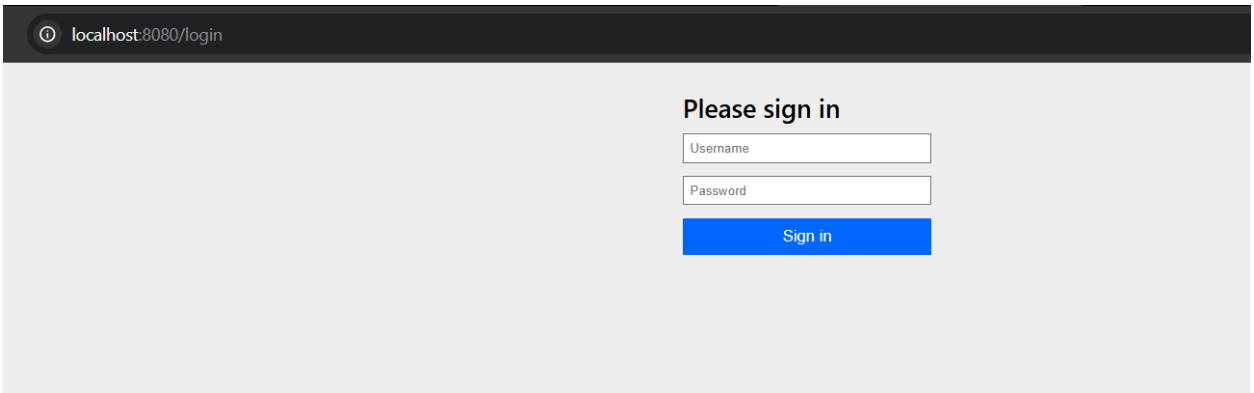
### 8. Application Controllers

- Once authenticated, the request proceeds to the application controllers to handle the business logic and return a response.

### 9. Response

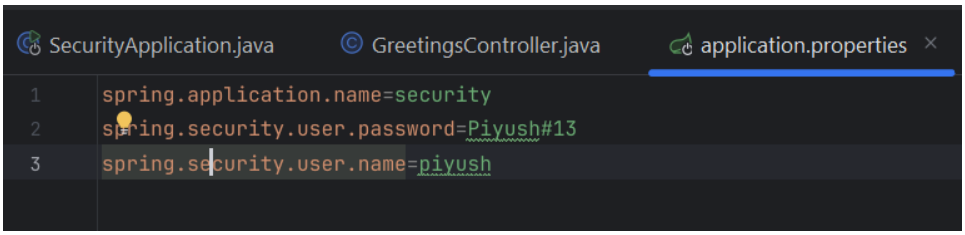
- The response is sent back to the user after passing back through the filter chain.

## If you run the project with spring security dependencies:




Now the endpoints will be protected

You can set ur own username and password as well




## Writing our own security filter

By default we have form based authentication we can change that lets see how

Default filter:-

```java
class SpringBootWebSecurityConfiguration {
    @Configuration(
            proxyBeanMethods = false
    )
    @ConditionalOnDefaultWebSecurity
    static class SecurityFilterChainConfiguration {
        SecurityFilterChainConfiguration() {
        }

        @Bean
        @Order(2147483642)
        SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
            http.authorizeHttpRequests(( AuthorizationManagerRequestMat... requests) -> ((AuthorizeHttpRequestsConfigurer.AuthorizedUrl) requests.ar
            http.formLogin(Customizer.withDefaults());
            http.httpBasic(Customizer.withDefaults());
            return (SecurityFilterChain) http.build();
        }
    }

    @Configuration(
            proxyBeanMethods = false
    )
    @ConditionalOnMissingBean(
            name = {"springSecurityFilterChain"}
    )
    @ConditionalOnClass({EnableWebSecurity.class})
    @EnableWebSecurity
    static class WebSecurityEnablerConfiguration {
        WebSecurityEnablerConfiguration() {
```

## Custom filter

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    //we are returning a filter chain hence func type is that and argument is of httpsecurity type (remember these two things)
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(( AuthorizationManagerRequestMat... requests) -> ((AuthorizeHttpRequestsConfigurer.AuthorizedUrl) reque
        //http.formLogin(Customizer.withDefaults());
        http.httpBasic(Customizer.withDefaults());
        return (SecurityFilterChain) http.build();
    }
}
```

@configuration-tells spring that this class provides configuration

@EnableWevSecurity-enables web security in our class

## In memory authentication(with database)

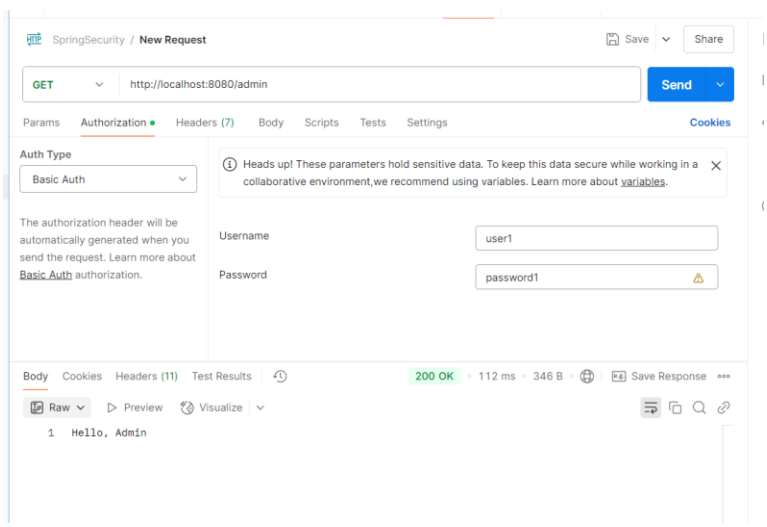
Creating new users within the app itself

```java
@Bean
//its predefine function --> UserDetailsService
public UserDetailsService userDetailsService() {
    //theres a class UserDetails thats used to deign user details
    UserDetails user1= User.withUsername("user1")
            .password("{noop}password1") //noop-->to save password as plain text
            .roles("USER")
            .build(); //created a user here itself
    UserDetails admin= User.withUsername("admin")
            .password("{noop}password2") //noop-->to encode ur password
            .roles("ADMIN")
            .build();
    return new InMemoryUserDetailsManager(user1,admin); //it manages user details in memory
}
```

## Role based authentication

Rn we can access by user1 and admin these two that we created that's it

People with different roles can different access over the details like password and all




```java
@GetMapping("admin")
public String adminEndPoint(){
    return "Hello, Admin";
}

@GetMapping("user")
public String userEndPoint(){
    return "Hello, user";
}
```

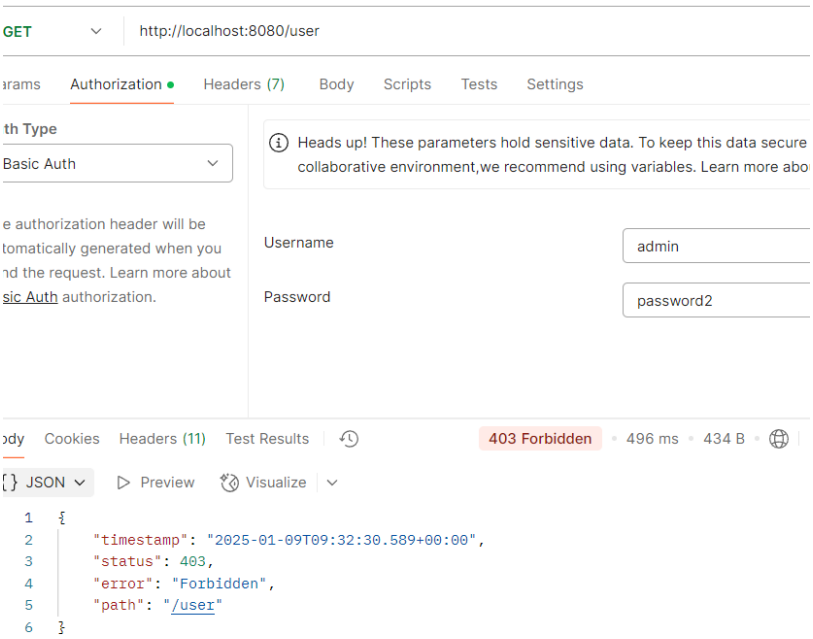
We are entering user credentials and getting admin data we don't want that that's what role based authorization is about

User must have only certain roles and admin must have others everyone cannot have all roles

```
@PreAuthorize("hasRole('USER')")
@GetMapping("user")
public String userEndPoint(){
    return "Hello, user";
}
```

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig
```

This will fix it




now admin cant access user url its forbidden

1.@PreAuthorize and @EnableMethodSecurity enables role based access

## Enabling H2 database

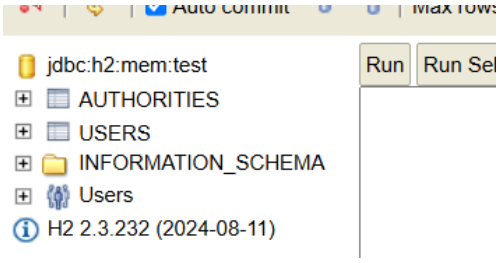
Till now we had in memry authentication but that wont happen mostl and we would have t make use of data base in real projects so lets see how there authentication works

```
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:test
```

```
                .anyRequest()).authenticated()); //if this h2 regust appears allow everyon no login page neede
http.sessionManagement( SessionManagementConfigurer<HttpSecurity> session->
                    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
 //http.formLogin(Customizer.withDefaults());
//toenable the framse if u dont see h2
http.headers( HeadersConfigurer<HttpSecurity> headers->headers.frameOptions( FrameOptionsConfig frameOptions->frameOptions.sameOrigin()));
http.csrf( CsrfConfigurer<HttpSecurity> csrf-> csrf.disable());
http.httpBasic(Customizer.withDefaults());

return (SecurityFilterChain) http.build();
```

## Database Authentication

- Till now we are using in memory user and admin now we will use the database to fetch user details for authentication
- We will be using JDBC user details manager
- Userdetailsservice is used to create user in memoy whereas Jdbcuserdetailmanager is used to create user in database
- 
```
        .build();

JdbcUserDetailsManager userDetailsManager = new JdbcUserDetailsManager(dataSource);
userDetailsManager.createUser(user1);
userDetailsManager.createUser(admin);
return userDetailsManager;
//return new InMemoryUserDetailsManager(user1,admin); //it manages user details in memory
```
- But this gies error that table users is not found this is coz for this method we have to define a schema for the user
- In the previous case UserDetails comes with a  schema itself that's hy it was not needed then but its needed now…
- JDBC (Java Database Connectivity) is a Java-based API (Application Programming Interface) that enables Java applications to interact with databases. It provides a standard set of methods and interfaces for connecting to a database, sending SQL queries, and retrieving results.
- To create a schema create a schema file in resources and add the schema there
- 
```
plication.java   GreetingsController.java   SecurityConfig.java   schema.sql   application.properties   pom.xml (security)

create table users(username varchar_ignorecase(50) not null primary key,password varchar_ignorecase(500) not null,enabled
create table authorities (username varchar_ignorecase(50) not null,authority varchar_ignorecase(50) not null,constraint fk
create unique index ix_auth_username on authorities (username,authority);
```
- So JDBC plus schema in resources will connect u to database that's it




SELECT * FROM USERS;

| USERNAME | PASSWORD | ENABLED |
|---|---|---|
| user1 | {noop}password1 | TRUE |
| admin | {noop}password2 | TRUE |

(2 rows, 9 ms)

Edit

- From in memory we reached here but still password is visible thasts not good

## Hashing

Hashing is the process of changing a simple string to something like encrypted message we cant password as it is that's not right.

- Hashing involves algorithms
- One of this algo is bcrypt that involves salting
- Salting adds additional layer of security
- It adds an additional string to ur program or ur data or password

SELECT * FROM USERS;

| USERNAME | PASSWORD | ENABLED |
|---|---|---|
| user1 | $2a$10$xDrqjNnUkZ5f9Yup/spqoOGM1N4M49U058JkLY0C3aZhKat3Lv3PW | TRUE |
| admin | $2a$10$cl/kwr.ImJQphhjW9hBJzeZHiiVR..PIho2yJsAG4UnbHvGkbw1OO | TRUE |

(2 rows, 6 ms)

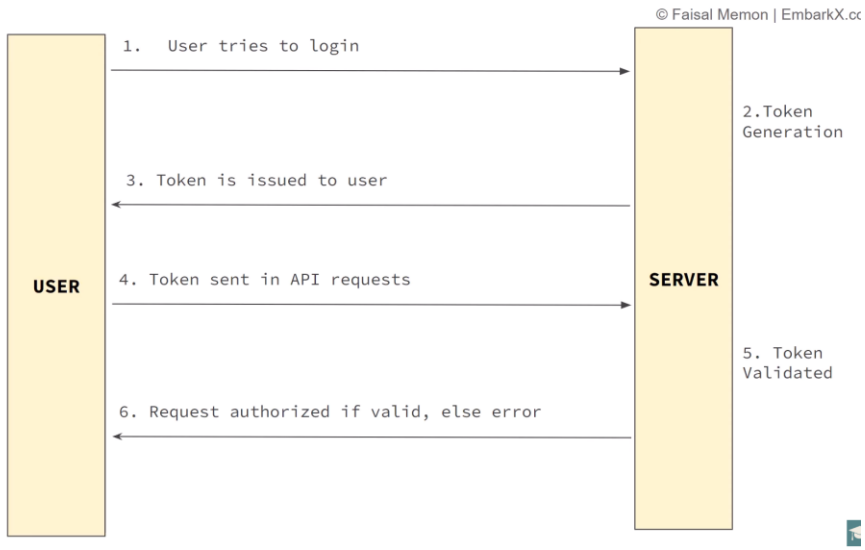
```
//theres a class UserDetails thats used to deign user details
UserDetails user1= User.withUsername("user1")
        .password(passwordEncoder().encode( rawPassword: "password1")) //noop-->to save password as plain tex
        .roles("USER")
        .build(); //created a user here itself
UserDetails admin= User.withUsername("admin")
        .password(passwordEncoder().encode( rawPassword: "password2")) //noop-->to encode ur password
        .roles("ADMIN")
        .build();

JdbcUserDetailsManager userDetailsManager = new JdbcUserDetailsManager(dataSource);
userDetailsManager.createUser(user1);
userDetailsManager.createUser(admin);
return userDetailsManager;
//return new InMemoryUserDetailsManager(user1,admin); //it manages user details in memory


blic PasswordEncoder passwordEncoder() { 2 usages
 return new BCryptPasswordEncoder();
```

# JWT Authentication(Jason Web Token)

## How things work without JWT

No advanced features like expiration time

Can be decoded easily

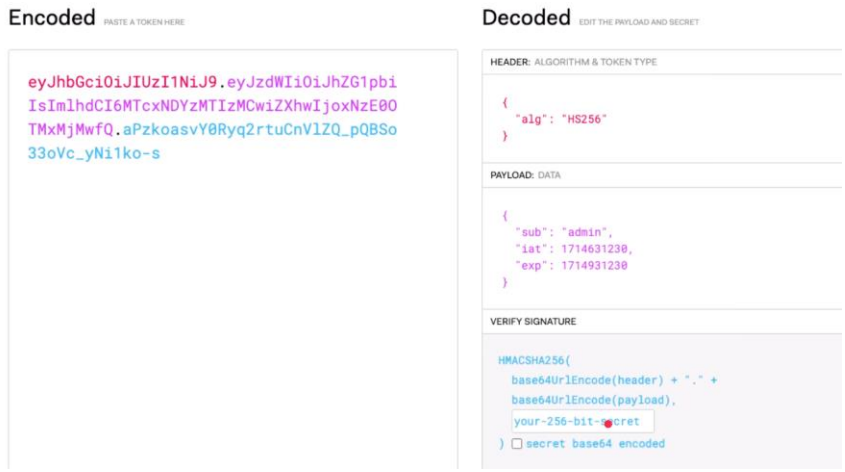



Uer

- User sends his username and password
- Spring Authenticates and generates a token for user
- Now whenever user makes an api call to any endpoint server first checks if token Is valid and if yes it send the response back to user.

# Heres how a token looks like

It has 3 parts header, data and erif signature




Data is there in payload

## Implementation of JWT

- WE will be creating three main files
- 1. Jwt utils 2. AuthTokenFilter 3. AuthEntryPoint
- 

**JwtUtils**

→ Contains utility methods for generating, parsing, and validating JWTs.

→Include generating a token from a username, validating a JWT, and extracting the username from a token.

**AuthTokenFilter**

→ Filters incoming requests to check for a valid JWT in the header, setting the authentication context if the token is valid.

→Extracts JWT from request header, validates it, and configures the Spring Security context with user details if the token is valid.

**AuthEntryPointJwt**

→ Provides custom handling for unauthorized requests, typically when authentication is required but not supplied or valid.

→When an unauthorized request is detected, it logs the error and returns a JSON response with an error message, status code, and the path attempted.

**SecurityConfig**

→ Configures Spring Security filters and rules for the application

→Sets up the security filter chain, permitting or denying access based on paths and roles. It also configures session management to stateless, which is crucial for JWT usage.

The authtokn filter is custom fillter and hence it requires a blue prit and that is security config

All throughout the authenticaion This security config is considerd

## Implementation

1. Insert the dependencies first that ull fing in jwt github repo

github.com/jwtk/jjwt?tab=readme-ov-file#maven

README License Security

Maven

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.6</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
<!-- Uncomment this next dependency if you are using:
```

# JWTUtils code

```
//JWT utils functions are 1. generating token from username 2. validating jwt tokens 3.Extrating username from token
@Component  1 usage
public class JwtUtils {
    private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);  5 usages
    //
    @Value("${spring.app.jwtSecret}")
    private String jwtSecret;

    @Value("${spring.app.jwtExpirationMs}")
    private int jwtExpirationMs;

    //Getting jwt token from header
    public String getJwtFromHeader(HttpServletRequest request) {  no usages
        String bearerToken = request.getHeader( s: "Authorization");
        logger.debug("Authorization Header: {}", bearerToken);
        if (bearerToken != null && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring( beginIndex: 7); // Remove Bearer prefix
        }// if it starts with bearer_ take the substring from 7th index
        return null;
    }

    //Generating token from username
    public String generateTokenFromUsername(UserDetails userDetails) {  no usages
        String username = userDetails.getUsername();
        return Jwts.builder()
                .subject(username)
                .issuedAt(new Date())
                .expiration(new Date((new Date()).getTime() + jwtExpirationMs))
                .signWith(key())
                .compact();
    }
```

```java
    //Extrating username from jwt token
    public String getUserNameFromJwtToken(String token) {  no usages
        return Jwts.parser() JwtParserBuilder
                .verifyWith((SecretKey) key())
                .build().parseSignedClaims(token) Jws<Claims>
                .getPayload().getSubject(); //payoad has the data
    }

    private Key key() {  3 usages
        return Keys.hmacShaKeyFor(Decoders.BASE64.decode(jwtSecret));
    }

    public boolean validateJwtToken(String authToken) {  no usages
        try {
            System.out.println("Validate");
            Jwts.parser().verifyWith((SecretKey) key()).build().parseSignedClaims(authToken);
            return true;
        } catch (MalformedJwtException e) {
            logger.error("Invalid JWT token: {}", e.getMessage());
        } catch (ExpiredJwtException e) {
            logger.error("JWT token is expired: {}", e.getMessage());
        } catch (UnsupportedJwtException e) {
            logger.error("JWT token is unsupported: {}", e.getMessage());
        } catch (IllegalArgumentException e) {
            logger.error("JWT claims string is empty: {}", e.getMessage());
        }
        return false;
    }
```

## 2.AuthTokenFilter

```java
@Component
public class AuthTokenFilter extends OncePerRequestFilter {
    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private UserDetailsService userDetailsService;

    private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);  4 usages

    @Override  no usages
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        logger.debug("AuthTokenFilter called for URI: {}", request.getRequestURI());
        try {
            String jwt = parseJwt(request); //first get the jwt
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                //gt username and get userdetails using username
                String username = jwtUtils.getUserNameFromJwtToken(jwt);

                UserDetails userDetails = userDetailsService.loadUserByUsername(username);

                UsernamePasswordAuthenticationToken authentication =
                        new UsernamePasswordAuthenticationToken(userDetails,
                                credentials: null,
                                userDetails.getAuthorities());
                logger.debug("Roles from JWT: {}", userDetails.getAuthorities());

                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
```

```java
    } catch (Exception e) {
        logger.error("Cannot set user authentication: {}", e);
    }

    filterChain.doFilter(request, response);
}

private String parseJwt(HttpServletRequest request) {  1 usage
    String jwt = jwtUtils.getJwtFromHeader(request);
    logger.debug("AuthTokenFilter.java: {}", jwt);
    return jwt;
}
```

The `AuthTokenFilter` class is a Spring Security filter that processes incoming HTTP requests to validate and authenticate a user based on a JWT (JSON Web Token). Here's a breakdown of the steps involved in the `doFilterInternal` method:

### Steps Involved:

1. **Extract JWT from the Request Header**:
   - The `parseJwt` method retrieves the JWT from the request's `Authorization` header using a utility function (`jwtUtils.getJwtFromHeader`).
2. **Validate the JWT**:
   - If a JWT is present, it is validated using the `jwtUtils.validateJwtToken` method to ensure it's not expired, malformed, or tampered with.
3. **Retrieve the Username from JWT**:
   - After successful validation, the username is extracted from the token using `jwtUtils.getUserNameFromJwtToken`.
4. **Load User Details**:
   - The `UserDetailsService` is used to fetch the `UserDetails` object for the username extracted from the JWT. This includes roles, permissions, and other user-related information.
5. **Create an Authentication Object**:
   - A `UsernamePasswordAuthenticationToken` is created using the `UserDetails`. This object contains the user's identity and their granted authorities (roles/permissions).
6. **Set Authentication in Security Context**:
   - The authentication object is set into the `SecurityContextHolder`, which holds the security context for the current thread of execution. This effectively authenticates the user for the request.
7. **Continue the Filter Chain**:
   - Once the user is authenticated, the request proceeds to the next filter or endpoint via `filterChain.doFilter`.
8. **Error Handling**:
   - Any exceptions during processing (e.g., invalid token, user not found) are logged using the `Logger`.

By implementing these steps, the filter ensures that every request is authenticated if it carries a valid JWT, allowing secure access to protected resources.

## 3.AuthEntryPointJwt

```java
@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt.class);  1 usage

    @Override  no usages
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)
            throws IOException, ServletException {
        logger.error("Unauthorized error: {}", authException.getMessage());

        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        final Map<String, Object> body = new HashMap<>();
        body.put("status", HttpServletResponse.SC_UNAUTHORIZED);
        body.put("error", "Unauthorized");
        body.put("message", authException.getMessage());
        body.put("path", request.getServletPath());

        final ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), body);
    }
```

## Sign in FLOW

```java
public class GreetingsController {
    @PostMapping("/signin")  new *
    public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest loginRequest) {
        Authentication authentication;
        try {
            authentication = authenticationManager
                    .authenticate(new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));
            //create a token using usernamePassword.... and then authenticate it
        } catch (AuthenticationException exception) {
            Map<String, Object> map = new HashMap<>();
            map.put("message", "Bad credentials");
            map.put("status", false);
            return new ResponseEntity<Object>(map, HttpStatus.NOT_FOUND);
        }

        //important note abt context add this always
        SecurityContextHolder.getContext().setAuthentication(authentication);

        UserDetails userDetails = (UserDetails) authentication.getPrincipal();

        String jwtToken = jwtUtils.generateTokenFromUsername(userDetails);

        List<String> roles = userDetails.getAuthorities().stream()  Stream<capture of extends GrantedAuthority>
                .map( capture of extends GrantedAuthority item -> item.getAuthority())  Stream<String>
                .collect(Collectors.toList());

        LoginResponse response = new LoginResponse(userDetails.getUsername(), roles, jwtToken);

        return ResponseEntity.ok(response);
    }
```

1. **Receive Login Request:** `LoginRequest` (username, password) is taken via `@RequestBody`. //theres separatee class defining structure of loginRequest
2. **Authenticate Credentials:**
   - `AuthenticationManager` validates credentials using `UsernamePasswordAuthenticationToken`.
   - If invalid, return "Bad credentials" with `HttpStatus.NOT_FOUND`.
3. **Set Security Context:**
   - On success, store `Authentication` in `SecurityContextHolder`.
4. **Generate JWT Token:**
   - Extract `username` from `UserDetails` and generate a JWT token.
5. **Retrieve Roles:**
   - Fetch roles/authorities of the user from `UserDetails`.
6. **Build Response:**
   - Create `LoginResponse` with `username`, `roles`, and `jwtToken`.
   - Return the response (`ResponseEntity.ok`).

# IMP

- **User sends `POST /signin` request with `username` and `password`.**
- **Credentials are authenticated using `AuthenticationManager`.**

  - **If valid: Proceed.**
  - **If invalid: Return "Bad credentials" with `HttpStatus.NOT_FOUND`.**

- **Set the authenticated user in the Spring Security Context.**
- **Generate a JWT token for the user.**
- **Retrieve user roles from the `UserDetails`.**

- **Return a `LoginResponse` containing the username, roles, and JWT token.**

**This ensures authentication, security context setup, and token generation for further API calls**.

# Managing Security Configurations

```java
public class SecurityConfig {
    private AuthEntryPointJwt unauthorizedHandler;

    @Bean  new *
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }

    @Bean  Piyush Singh *
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests( AuthorizationManagerRequestMat... authorizeRequests ->
                authorizeRequests.requestMatchers("/h2-console/**").permitAll()
                        .requestMatchers("/signin").permitAll()
                        .anyRequest().authenticated());
        http.sessionManagement(
                SessionManagementConfigurer<HttpSecurity> session ->
                        session.sessionCreationPolicy(
                                SessionCreationPolicy.STATELESS)
        );
        http.exceptionHandling( ExceptionHandlingConfigurer<HttpSecurity> exception -> exception.authenticationEntryPoint(unauthorizedHandler));
        //http.httpBasic(withDefaults());
        http.headers( HeadersConfigurer<HttpSecurity> headers -> headers
                .frameOptions( FrameOptionsConfig frameOptions -> frameOptions
                        .sameOrigin()
                )
        );
        http.csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable());
        http.addFilterBefore(authenticationJwtTokenFilter(),
                UsernamePasswordAuthenticationFilter.class);
```

## 1. Authorize Requests

```
http.authorizeHttpRequests(authorizeRequests ->
    authorizeRequests.requestMatchers("/h2-console/**").permitAll()
                     .requestMatchers("/signin").permitAll()
                     .anyRequest().authenticated());
```

Defines access control rules for different endpoints:

- `/h2-console/**` and `/signin` are marked as public, meaning anyone can access them without authentication.
- `anyRequest().authenticated()` ensures that all other endpoints require the user to be authenticated before access.

## 2. Stateless Session Management

```
http.sessionManagement(session ->
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

Configures the application to use stateless session management, ensuring the server does not store user sessions. Instead, each request must include authentication credentials, such as a valid JWT. This approach is common for APIs where the client manages the authentication token.

## 3. Exception Handling

```
http.exceptionHandling(exception ->
    exception.authenticationEntryPoint(unauthorizedHandler));
```

Specifies a custom `unauthorizedHandler` to handle unauthorized access. When a user attempts to access a protected resource without proper authentication, the handler generates a custom error response, typically in JSON format, with details like an error message and HTTP status code.

### 4. Frame Options for H2 Console

```
http.headers(headers -> headers
    .frameOptions(frameOptions -> frameOptions.sameOrigin()));
```

Adjusts the `X-Frame-Options` header to allow the H2 database console to be displayed in an iframe on the same domain. This is a necessary adjustment during development when using the H2 console embedded in the project.

### 5. Disable CSRF Protection

```
http.csrf(csrf -> csrf.disable());
```

Disables Cross-Site Request Forgery (CSRF) protection. CSRF is unnecessary for stateless APIs, as JWT tokens include cryptographic signatures that inherently prevent such attacks. This configuration is essential for simplifying the interaction between the client and server in stateless authentication models.

### 6. Add JWT Authentication Filter

```
http.addFilterBefore(authenticationJwtTokenFilter(),
                     UsernamePasswordAuthenticationFilter.class);
```

Registers a custom filter, `authenticationJwtTokenFilter`, to intercept incoming requests and validate the JWT token. This filter is added before the default `UsernamePasswordAuthenticationFilter` to ensure token validation occurs before any username/password-based authentication.

### 7. Build and Return Security Filter Chain

```
return http.build();
```

Finalizes and returns the configured `SecurityFilterChain`, which Spring Security uses to apply the defined security rules and filters to incoming HTTP requests.

---

This approach ensures a secure, stateless, and JWT-compatible configuration for a Spring Security setup.

```java
@Bean  new *
public CommandLineRunner initData(UserDetailsService userDetailsService) {
    return String[] args -> {
        JdbcUserDetailsManager manager = (JdbcUserDetailsManager) userDetailsService;
        UserDetails user1 = User.withUsername("user1")
                .password(passwordEncoder().encode( rawPassword: "password1"))
                .roles("USER")
                .build();
        UserDetails admin = User.withUsername("admin")
                //.password(passwordEncoder().encode("adminPass"))
                .password(passwordEncoder().encode( rawPassword: "adminPass"))
                .roles("ADMIN")
                .build();

        JdbcUserDetailsManager userDetailsManager = new JdbcUserDetailsManager(dataSource);
        userDetailsManager.createUser(user1);
        userDetailsManager.createUser(admin);
    };
}
```

### Purpose of `initData` Method:

The `initData` method initializes user data in the database when the application starts. It creates default user accounts (e.g., `user1` and `admin`) for authentication and authorization. This is especially useful in development or testing environments.

---

### Detailed Breakdown:

#### Declaring the Method

```
public CommandLineRunner initData(UserDetailsService userDetailsService) {
```

**`CommandLineRunner`**: Runs logic after the application context is loaded.

**`UserDetailsService`**: Retrieves user-related data and is used to manage user details.

#### Returning a Lambda Function

```
return args -> {
```

This lambda function contains the logic for creating and storing user details during startup.

#### Casting `UserDetailsService` to `JdbcUserDetailsManager`

```
JdbcUserDetailsManager manager = (JdbcUserDetailsManager)
userDetailsService;
```

**`JdbcUserDetailsManager`**: A `UserDetailsService` implementation that stores and retrieves user details from a database.

### Defining User Details

```
UserDetails user1 = User.withUsername("user1")
        .password(passwordEncoder().encode("password1"))
        .roles("USER")
        .build();
UserDetails admin = User.withUsername("admin")
        .password(passwordEncoder().encode("adminPass"))
        .roles("ADMIN")
        .build();
```

**`UserDetails`**: Represents a user in Spring Security.

Creates two users:

`user1`: Username `user1`, password `password1`, role `USER`.

`admin`: Username `admin`, password `adminPass`, role `ADMIN`.

**Password Encoding**: Uses `PasswordEncoder` (e.g., `BCryptPasswordEncoder`) to hash passwords.

### Creating and Storing Users

```
JdbcUserDetailsManager userDetailsManager = new
JdbcUserDetailsManager(dataSource);
userDetailsManager.createUser(user1);
userDetailsManager.createUser(admin);
```

**`JdbcUserDetailsManager`**: Manages user details with JDBC.

**`createUser`**: Inserts user data into tables like `users` and `authorities`.

---

## Use Cases:

**Development/Testing**: Preloads default users for easier testing without manual data entry.

**Initial Setup**: Creates default admin/system accounts for a new environment.

**Learning/Prototyping**: Demonstrates how Spring Security handles authentication using JDBC.

---

## Notes:

**Production Security**: Remove such initialization logic in production or ensure default credentials are secured.

**Password Storage**: Always hash passwords with a secure `PasswordEncoder` before saving.

# Working fine