

Spring Security Configuration



```
package me.learning.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class AppConfig {

    // spring issi ke help se fetch karega user name and password jab security lagana hoga to
    // ye bean to fetch karega wo
    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails userDetails = User.builder()
            .username("Piyush")
            .password(passwordEncoder().encode("Piyush"))
            .build();

        // aur ek user banayenge agar to niche inmemoryuserdetailsmanager mein pass kr skte hai
        // usme var arguments hai to kitna bhi user pass kr skte hia
        return new InMemoryUserDetailsManager(userDetails);
    }

    // to encode the password
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

} explain each and everything in details so that i understand the flow
```

Sure Piyush! Let's go through your Spring Security configuration class line by line in **very simple language**, so that you understand **what it does**, **why it is needed**, and **how it fits into the Spring Security flow**.

✓ Class Overview

```
java
```

```
@Configuration
public class AppConfig {
```

- `@Configuration` tells Spring that this class contains **beans** (objects managed by Spring container).
- `AppConfig` is your custom configuration class for Spring Security settings like user credentials and password encoding.

✓ UserDetailsService Bean

```
java
```

```
@Bean
public UserDetails userDetailsService(){
```

- `@Bean`: This method creates a bean (i.e., reusable object) managed by Spring.
- Spring Security uses this bean to **fetch user details** like username and password at login time.
- `UserDetailsService` is an interface that Spring uses to retrieve user info for authentication.

✓ Creating UserDetails

```
java
```

```
UserDetails userDetails = User.builder()
    .username("Piyush")
    .password(passwordEncoder().encode("Piyush"))
    .build();
```

- `User.builder()`: Creates a user using the builder pattern.
- `.username("Piyush")`: You are creating a user with username = **Piyush**.
- `.password(...)`: You are giving the user a password, but encoding it using **BCrypt** (this is important because Spring Security does not allow plain text passwords).
- `passwordEncoder().encode("Piyush")`: This line encodes the password **Piyush** using the `BCryptPasswordEncoder` (see below).
- `.build()`: Finalizes the creation of the `UserDetails` object.



Why encode the password?

Spring Security compares the **encoded password** from the database (or in-memory) with the **encoded version of the password** you entered during login.

✓ Returning InMemoryUserDetailsManager

```
java
```

```
return new InMemoryUserDetailsManager(userDetails);
```

- This line returns an instance of `InMemoryUserDetailsManager` that holds the user(s) you've created.
- `InMemoryUserDetailsManager` is a **built-in Spring Security class** used to store users in memory (not in DB).
- You can pass multiple users to this class, like:

```
java

new InMemoryUserDetailsManager(user1, user2, user3);
```

✓ PasswordEncoder Bean

```
java

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- This defines a `PasswordEncoder` bean using `BCryptPasswordEncoder`, which is one of the **most secure hashing algorithms**.
- Spring will use this encoder whenever it needs to **match passwords** during login.

🧠 How the flow works when user logs in:

1. 🗝️ You have added **Spring Security** to your project.
2. Spring detects this configuration class and registers:
 - `UserDetailsService` → to fetch the user.
 - `PasswordEncoder` → to check password.
3. When someone logs in:
 - Spring calls `userDetailsService()` → gets **Piyush** user from memory.
 - Takes password entered by user (say "Piyush") → encodes it using `BCryptPasswordEncoder`.
 - Compares it with the stored encoded password.
 - ✓ If matched → login success.
 - ✗ If not matched → login failed.

💡 Important Points

Component	Purpose
<code>UserDetailsService</code>	Retrieves user info (username, encoded password)
<code>UserDetails</code>	Represents individual user info
<code>PasswordEncoder</code>	Encodes and compares passwords securely
<code>InMemoryUserDetailsManager</code>	Stores users in memory for testing or simple setups

✓ Example: Add one more user

If you want to add another user (for example, `admin`), you can modify the method like this:

```
java

@Bean
public UserDetailsService userDetailsService(){
    UserDetails user1 = User.builder()
        .username("Piyush")
        .password(passwordEncoder().encode("Piyush"))
        .roles("USER")
        .build();

    UserDetails user2 = User.builder()
        .username("admin")
        .password(passwordEncoder().encode("admin123"))
        .roles("ADMIN")
        .build();

    return new InMemoryUserDetailsManager(user1, user2);
}
```

Let me know if you want me to **show how to use these users in login form or REST APIs**, or explain **how role-based access control (ROLE_USER, ROLE_ADMIN)** works in Spring Security.



```
package me.learning.config;

import me.learning.security.JwtAuthenticationEntryPoint;
import me.learning.security.JwtAuthenticationFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.beans.factory.annotation.Autowired;

@Configuration
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationEntryPoint point;

    @Autowired
    private JwtAuthenticationFilter filter;
```

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/test").authenticated()
            .requestMatchers("/auth/login", "/auth/register").permitAll()
            .anyRequest().authenticated()
        )
        .exceptionHandling(ex -> ex
            .authenticationEntryPoint(point)
        )
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        );

    http.addFilterBefore(filter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}

@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
    return config.getAuthenticationManager();
}
} explain each and everything in details so that i understand the flow
```

Great, Piyush! You're now working on a **JWT-based Spring Security configuration** — this is a very important and modern approach used in real-world applications.

Let's break this down **line by line**, in **very simple language**, and explain how this setup works when a user logs in or accesses a secured API.

✓ Class Overview

```
java

@Configuration
public class SecurityConfig {
```

- Marks this class as a **Spring configuration** class.
- It defines **beans** for Spring Security — particularly how requests are authenticated and authorized using **JWT** (JSON Web Token).

✓ Autowired Components

```
java

@Autowired
private JwtAuthenticationEntryPoint point;

@Autowired
private JwtAuthenticationFilter filter;
```

◆ JwtAuthenticationEntryPoint

- This is a **custom class** (you created it) that handles what should happen when an **unauthenticated user** tries to access a protected endpoint.
- It sends a response like **401 Unauthorized** with a message like "Access denied".

◆ JwtAuthenticationFilter

- This is a **custom filter** that checks for JWT token in the **request header** (usually **Authorization: Bearer <token>**).
- If the token is valid, it sets the **authentication** inside the **SecurityContextHolder** so Spring knows the user is authenticated.

✓ SecurityFilterChain Bean

```
java

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
```

- This bean defines the **core security configuration** using the **HttpSecurity** object.
- This tells Spring how to **handle incoming HTTP requests**, whether to allow or block them.

✓ Step-by-step Configuration Inside securityFilterChain

1 Disable CSRF

```
java

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

- CSRF (Cross Site Request Forgery) is used in session-based authentication (like browser forms).
- Since you're using **stateless JWT-based auth**, you don't need **CSRF**, so it's disabled.

2 Authorize Requests

```
java

.authorizeHttpRequests(auth -> auth
    .requestMatchers("/test").authenticated()
    .requestMatchers("/auth/login", "/auth/register").permitAll()
    .anyRequest().authenticated()
)
```

- `.requestMatchers("/test").authenticated()`:
 - `/test` is a protected API.
 - Only authenticated users with a valid token can access it.
- `.requestMatchers("/auth/login", "/auth/register").permitAll()`:
 - These endpoints are public (no login required).
 - Used for logging in and registering users.
- `.anyRequest().authenticated()`:
 - Any other request should also be authenticated (unless explicitly allowed above).

3 Handle Unauthorized Access

```
java

.exceptionHandling(ex -> ex.authenticationEntryPoint(point))
```

- This tells Spring to use your custom `JwtAuthenticationEntryPoint` when a request is **unauthenticated or fails** authorization.

4 Session Management

```
java

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

- You're using **JWTs**, so you don't want to use HTTP sessions.
- `SessionCreationPolicy.STATELESS` ensures Spring Security does not create or use any session.

5 Add Custom JWT Filter

```
java

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

- You are adding your **custom filter** (`JwtAuthenticationFilter`) **before** Spring's built-in `UsernamePasswordAuthenticationFilter`.

- This is important because your filter checks the **JWT token** in the header before Spring tries to check username/password.
- It does this:
 - Reads token
 - Validates it
 - Loads user details
 - Sets authentication into Spring's **SecurityContextHolder**

Return Built **SecurityFilterChain**

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

- Builds and returns the **SecurityFilterChain** bean.

AuthenticationManager Bean

```
java
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws
Exception {
    return config.getAuthenticationManager();
}
```

- **AuthenticationManager** is a core part of Spring Security.
- Used to **authenticate username/password** when someone tries to log in (typically during **/auth/login**).
- You get it from **AuthenticationConfiguration**, which already wires everything.

Full Flow of JWT Security

Here's how it works from the moment a user tries to access your application:

Login Flow (/auth/login)

1. User sends **username and password**.
2. You use **AuthenticationManager** to **authenticate the user**.
3. If correct:
 - Generate a **JWT token**
 - Send it back in the response
4. Client stores the token (typically in **localStorage** or **sessionStorage** in frontend).

📄 Secured API Flow (like /test)

1. User sends a request with **Authorization header**:

```
makefile

Authorization: Bearer <JWT_TOKEN>
```

2. `JwtAuthenticationFilter` intercepts the request:

- Checks token validity
- Loads user from token
- Sets user authentication in the security context

3. If valid → allow access

4. If invalid or no token → `JwtAuthenticationEntryPoint` returns **401 Unauthorized**

✅ Summary Table

Component	Role
<code>JwtAuthenticationEntryPoint</code>	Handles unauthorized access
<code>JwtAuthenticationFilter</code>	Extracts, validates JWT from headers
<code>SecurityFilterChain</code>	Main security config for endpoints
<code>AuthenticationManager</code>	Authenticates credentials (e.g., during login)
<code>SessionCreationPolicy.STATELESS</code>	No session, use JWT for all auth
<code>.addFilterBefore(...)</code>	Makes sure your filter runs early in the filter chain

Would you like a **visual flow diagram** for this or a **template for the `JwtAuthenticationFilter` and `JwtAuthenticationEntryPoint` classes** as well?



```
package me.learning.controller;

import me.learning.model.JwtRequest;
import me.learning.model.JwtResponse;
import me.learning.security.JwtHelper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private AuthenticationManager manager;

    @Autowired
    private JwtHelper helper;

    @PostMapping("/login")
    public ResponseEntity<JwtResponse> login(@RequestBody JwtRequest request) {

        // Authenticate user
        this.doAuthenticate(request.getEmail(), request.getPassword());

        // Load user details
        UserDetails userDetails = userDetailsService.loadUserByUsername(request.getEmail());

        // Generate token
        String token = this.helper.generateToken(userDetails);

        // Build response
        JwtResponse response = JwtResponse.builder()
            .jwtToken(token)
            .username(userDetails.getUsername())
            .build();

        return new ResponseEntity<>(response, HttpStatus.OK);
    }

    private void doAuthenticate(String email, String password) {
        UsernamePasswordAuthenticationToken authentication =
            new UsernamePasswordAuthenticationToken(email, password);

        try {
            manager.authenticate(authentication);
        } catch (BadCredentialsException e) {
            throw new RuntimeException("Invalid Username or Password!!!");
        }
    }
}
```

explain each and everything in details so that i understand the flow

Great job, Piyush! You're now working on the **JWT login controller** — the part that handles login and generates JWT tokens. Let me explain everything step by step in **very simple and clear language**, so you understand exactly how it works.

✓ Class Overview

```
java

@RestController
@RequestMapping("/auth")
public class AuthController {
```

- **@RestController**: Marks this class as a **Spring REST controller**. It handles HTTP requests and returns JSON responses.
- **@RequestMapping("/auth")**: All APIs in this class will start with **/auth**, like:
 - **/auth/login**

🔧 Autowired Dependencies

1 UserDetailsService userDetailsService

```
java

@Autowired
private UserDetailsService userDetailsService;
```

- Spring's interface used to **load user details** (like username, password, roles).
- Used here to get the **UserDetails** object of the user after authentication.

2 AuthenticationManager manager

```
java

@Autowired
private AuthenticationManager manager;
```

- Spring Security's built-in class to handle the **actual authentication**.
- Takes a **UsernamePasswordAuthenticationToken**, checks the username and password.

3 JwtHelper helper

```
java

@Autowired
```

```
private JwtHelper helper;
```

- Your **custom class** responsible for:
 - Generating JWT tokens (`generateToken(...)`)
 - Validating them (probably in your filter)
- You pass user details to this helper to **create a JWT**.

✓ Login API

```
java

@PostMapping("/login")
public ResponseEntity<JwtResponse> login(@RequestBody JwtRequest request)
```

- API endpoint: `/auth/login`
- Method: `POST`
- Request body: contains `email` and `password`, which comes from `JwtRequest` class.
- Response: returns a JSON response with JWT token and username (`JwtResponse`).

🔄 Step-by-step Inside `login()` method

1 Authenticate User

```
java

this.doAuthenticate(request.getEmail(), request.getPassword());
```

- Calls a private method to verify whether email & password are correct.

2 Load User Details

```
java

UserDetails userDetails = userDetailsService.loadUserByUsername(request.getEmail());
```

- Loads user from Spring's memory/database using the email.
- Returns a `UserDetails` object which contains:
 - Username
 - Encoded password
 - Roles (if any)

3 Generate Token

```
java

String token = this.helper.generateToken(userDetails);
```

- Uses your `JwtHelper` class to generate a JWT token using the user's details.
- JWT typically includes:
 - Subject (email/username)
 - Issue time
 - Expiry
 - Signature

4 Build Response

```
java

JwtResponse response = JwtResponse.builder()
    .jwtToken(token)
    .username(userDetails.getUsername())
    .build();
```

- Creates a response object with the generated token and username.
- `JwtResponse` is a POJO class with fields like `jwtToken`, `username`.

5 Return Response

```
java

return new ResponseEntity<>(response, HttpStatus.OK);
```

- Sends a response back to client with status 200 OK.
- JSON body includes the JWT token.

✓ Private Method: doAuthenticate()

```
java

private void doAuthenticate(String email, String password)
```

- Handles authentication manually using Spring's `AuthenticationManager`.

What it does:

```
java
```

```
UsernamePasswordAuthenticationToken authentication =
    new UsernamePasswordAuthenticationToken(email, password);
```

- Wraps the email and password into a Spring Security object (`UsernamePasswordAuthenticationToken`).

```
java
```

```
manager.authenticate(authentication);
```

- Tries to authenticate this user using `AuthenticationManager`.

If password is wrong:

```
java
```

```
} catch (BadCredentialsException e) {
    throw new RuntimeException("Invalid Username or Password!!");
}
```

- If the password or email is incorrect, Spring throws a `BadCredentialsException`.
- You catch this and throw a user-friendly error.

✓ JWT Login Flow: End-to-End

1. User sends login request:

```
pgsql
```

```
POST /auth/login
Content-Type: application/json
```

```
{
  "email": "piyush",
  "password": "piyush"
}
```

2. Backend flow:

Step	Action
1	<code>AuthController.login()</code> is called
2	<code>doAuthenticate()</code> checks credentials
3	If valid → load user details
4	Generate JWT token using <code>JwtHelper</code>
5	Return token in <code>JwtResponse</code>

3. Response sent back:

```
json

{
  "jwtToken": "<JWT-TOKEN-HERE>",
  "username": "piyush"
}
```

✓ Summary Table

Component	Role
AuthenticationManager	Authenticates email & password
UserDetailsService	Loads user from memory/db
JwtHelper	Creates JWT token
JwtRequest	Contains login email & password
JwtResponse	Contains generated JWT token
UsernamePasswordAuthenticationToken	Spring's object for login credentials
BadCredentialsException	Thrown when login fails

Would you like:

- A visual diagram of the login flow?
- Or the code template for **JwtHelper**, **JwtRequest**, and **JwtResponse** classes?

Let me know!



```
package me.learning.security;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.io.PrintWriter;

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request,
                        HttpServletResponse response,
                        AuthenticationException authException) throws IOException {

        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
    }
}
```

```

    PrintWriter writer = response.getWriter();
    writer.println("Access Denied !! " + authException.getMessage());
}
} explain each and everything in details so that i understand the flow

```

Great, Piyush! This class `JwtAuthenticationEntryPoint` plays an important role in **handling unauthorized access** in your JWT-based Spring Security system.

Let's go through this **line by line**, in **very simple language**, and understand what happens behind the scenes.

✓ Purpose of `JwtAuthenticationEntryPoint`

This class is used by Spring Security when a **user who is *not authenticated*** tries to access a **secured endpoint**.

🔑 For example:

- The client tries to call `/test` **without providing a valid JWT token**, or with an **expired/invalid token**.
- Spring Security will **not allow** this request and will call this class to generate an appropriate response (like 401 Unauthorized).

✓ Class Declaration

```

java

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

```

- `@Component`: Registers this class as a Spring Bean so it can be auto-wired in your `SecurityConfig`.
- `implements AuthenticationEntryPoint`: You are **overriding a method** that tells Spring **what to do when authentication fails**.

✓ Overridden Method: `commence(...)`

```

java

@Override
public void commence(HttpServletRequest request,
                    HttpServletResponse response,
                    AuthenticationException authException) throws IOException {

```

- This method is called **automatically by Spring** when a user tries to access a **protected resource** and:
 - Is not authenticated

- Or provides a bad/missing JWT token

✓ Set HTTP Status Code

```
java

response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
```

- You tell Spring to return **HTTP 401 Unauthorized** status.
- This status code means: "You are not logged in or token is invalid."

✓ Send Error Message to Client

```
java

PrintWriter writer = response.getWriter();
writer.println("Access Denied !! " + authException.getMessage());
```

- You write a message into the response body.
- `authException.getMessage()` provides more details (e.g., "Full authentication is required").
- The client (like Postman or frontend app) will **see this message in the response**.

Real Example Flow

Let's say the `/test` API is protected:

```
java

.requestMatchers("/test").authenticated()
```

Case: User Sends No Token

```
http

GET /test
(no Authorization header)
```

Result:

1. Spring sees there's no JWT.
2. It knows this request must be authenticated.
3. Spring calls `JwtAuthenticationEntryPoint.commence()`.
4. This sets the response to:

```
http
```

```
HTTP/1.1 401 Unauthorized
Access Denied !! Full authentication is required to access this resource
```

✓ Where it is Used in Your App?

In your `SecurityConfig`:

```
java

.exceptionHandling(ex -> ex.authenticationEntryPoint(point))
```

- Here, you're telling Spring to **use your custom `JwtAuthenticationEntryPoint`** when there's an authentication error.

✓ Summary

Line of Code	What It Does
<code>implements AuthenticationEntryPoint</code>	Allows you to customize the response when authentication fails
<code>commence(...)</code>	Called automatically when an unauthenticated request hits a protected endpoint
<code>response.setStatus(401)</code>	Sends HTTP 401 (Unauthorized)
<code>response.getWriter().println(...)</code>	Writes an error message to the client
<code>@Component</code>	Makes this class available for auto-wiring in Spring

✓ Final Thoughts

- This class **only runs when token is missing or invalid**.
- If token is valid, Spring uses the filter and allows the request.
- If user is authenticated but **not authorized**, then a different class (`AccessDeniedHandler`) would handle it — let me know if you'd like to add that too.

Let me know if you'd like a visual of this flow or the next class (`JwtAuthenticationFilter`) explanation.



```
package me.learning.security;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
```

```
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.io.PrintWriter;

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request,
                        HttpServletResponse response,
                        AuthenticationException authException) throws IOException {

        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
        PrintWriter writer = response.getWriter();
        writer.println("Access Denied !! " + authException.getMessage());
    }
} explain each and everything in details so that i understand the flow
```

Absolutely, Piyush! You're looking at the class `JwtAuthenticationEntryPoint`, which is a **core part of JWT-based Spring Security**. It handles **what should happen when a request comes in without proper authentication**.

Let's go step-by-step and explain the **role, flow, and each line** so you understand it completely.

✓ Purpose of `JwtAuthenticationEntryPoint`

When someone accesses a protected API **without a valid token**, Spring Security says:

"Hey! This user isn't authenticated. What should I do?"

👉 That's where this class comes in — Spring delegates the response handling to this class.

It returns a **401 Unauthorized response** with a custom error message like:

```
pgsql

Access Denied !! Full authentication is required to access this resource
```

✓ Code Breakdown

📌 Annotations and Class Declaration

```
java

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {
```

- `@Component` → This tells Spring to **auto-detect this class as a bean**.
- Implements `AuthenticationEntryPoint` → This is a Spring Security interface used to **handle unauthorized access**.

✓ It is only triggered when the **user is not authenticated** — like missing or invalid token.

📌 Overridden Method: `commence(...)`

```
java

@Override
public void commence(HttpServletRequest request,
                     HttpServletResponse response,
                     AuthenticationException authException) throws IOException {
```

This method is **called automatically** when:

- A protected API is accessed
- But there's **no valid authentication**

Parameters:

- `HttpServletRequest request`: The actual HTTP request
- `HttpServletResponse response`: The object you will write your response to
- `AuthenticationException authException`: Exception thrown by Spring when authentication fails

📌 Set HTTP Status Code

```
java

response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
```

- You are saying: "This request is **unauthorized**, so return HTTP 401".
- `SC_UNAUTHORIZED = 401`

📌 Write a Custom Error Message to the Response Body

```
java

PrintWriter writer = response.getWriter();
writer.println("Access Denied !! " + authException.getMessage());
```

- `response.getWriter()` returns a `PrintWriter` to write response text.
- You print a message like:

pgsql

Access Denied !! Full authentication is required to access this resource

✓ Real-Life Example

📌 Suppose this is your protected API:

```
java
@GetMapping("/test")
public ResponseEntity<String> getTestData() {
    return ResponseEntity.ok("Hello, Piyush!");
}
```

📌 You call it with no token:

```
http

GET /test
(no Authorization header)
```

↻ What Happens:

1. Spring sees: This endpoint requires authentication
2. No valid token is present
3. Spring throws `AuthenticationException`
4. Spring calls your `JwtAuthenticationEntryPoint.commence()`
5. Your method sets:
 - Status = 401
 - Message = "Access Denied !! Full authentication is required to access this resource"

✓ When Is This Triggered?

Scenario	Will this class handle it?
No token sent	✓ Yes
Invalid/expired token	✓ Yes
User authenticated but no role	✗ No (that's handled by <code>AccessDeniedHandler</code>)

✓ Summary Table

Component	Role
@Component	Registers this class as a Spring-managed bean

Component	Role
AuthenticationEntryPoint	Interface for handling authentication errors
commence()	Called when a request is not authenticated
response.setStatus(401)	Sends HTTP 401 Unauthorized
response.getWriter().println(...)	Sends error message to the client

✓ Full Flow Overview

pgsql

Client → Makes request to protected API (/test)
 → No token (or invalid)
 → Spring Security fails authentication
 → JwtAuthenticationEntryPoint.commence() is triggered
 → Sends back 401 + "Access Denied" message

Would you like me to now explain the `JwtAuthenticationFilter` class and how it connects the JWT token to Spring's authentication context? That's the next important piece in this flow.



```
package me.learning.security;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Component
public class JwtHelper {

    // Token validity (5 hours in seconds)
    public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;

    // Secret key used to sign the token
    private String secret =
        "m9ZhQnhGsl8X2H1HGeXGc5bPmV9F2Vx9JrDxg9y0eB4Vc8Hr9kUgCy0RZQWxB9IW";

    // ✓ Retrieve username from JWT token
    public String getUsernameFromToken(String token) {
        return getClaimFromToken(token, Claims::getSubject);
    }
}
```

```
// ✅ Retrieve expiration date from JWT token
public Date getExpirationDateFromToken(String token) {
    return getClaimFromToken(token, Claims::getExpiration);
}

// ✅ Retrieve specific claim from JWT token
public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = getAllClaimsFromToken(token);
    return claimsResolver.apply(claims);
}

// ✅ Get all claims using the secret key
private Claims getAllClaimsFromToken(String token) {
    return Jwts.parser()
        .setSigningKey(secret)
        .parseClaimsJws(token)
        .getBody();
}

// ✅ Check if token is expired
private Boolean isTokenExpired(String token) {
    final Date expiration = getExpirationDateFromToken(token);
    return expiration.before(new Date());
}

// ✅ Generate JWT token for a given user
public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    return doGenerateToken(claims, userDetails.getUsername());
}

// 🛠️ Method that creates JWT token with claims
private String doGenerateToken(Map<String, Object> claims, String subject) {
    return Jwts.builder()
        .setClaims(claims) // claims like issuer, role, etc.
        .setSubject(subject) // username
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
        .signWith(SignatureAlgorithm.HS512, secret)
        .compact();
}

// ✅ Validate token
public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = getUsernameFromToken(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
} explain each and everything in details so that i understand the flow
```

Great job, Piyush! You've now reached the **heart of JWT handling** in Spring Security — your `JwtHelper` class is responsible for **generating**, **parsing**, and **validating** JWT tokens.

Let me explain this line by line in very **simple and clear language** so you understand the **entire flow of JWT operations**.

✓ What is JwtHelper?


It's a Spring `@Component` that provides utility methods to:

Function	Purpose
<code>generateToken()</code>	Create a JWT for a user
<code>getUsernameFromToken()</code>	Extract username (subject) from the token
<code>getExpirationDateFromToken()</code>	Get token expiry time
<code>validateToken()</code>	Check if token is valid (not expired + username match)

✓ Class-Level Constants

```
java

public static final long JWT_TOKEN_VALIDITY = 5 * 60 * 60;
private String secret = "m9ZhQnhGsL8X2H1HGeXGc5bPmV9F2Vx9JrDxg9y0eB4Vc8Hr9kUgCy0RZQWxB9lW";
```

- `JWT_TOKEN_VALIDITY`: Token is valid for **5 hours** = `5 * 60 * 60` seconds
- `secret`: Secret key to sign and parse JWT tokens.
 **Keep it safe** and long enough for `HS512`.

✓ Get Username from Token

```
java

public String getUsernameFromToken(String token) {
    return getClaimFromToken(token, Claims::getSubject);
}
```

- JWT stores the **username in the "subject" (sub) field**
- This method extracts that using the helper method `getClaimFromToken`

✓ Get Expiration Date from Token

```
java

public Date getExpirationDateFromToken(String token) {
    return getClaimFromToken(token, Claims::getExpiration);
}
```


- JWT contains an expiry time (**exp** field)
- This method helps you get the expiry time of a token

✓ Generic Method to Extract Any Claim

```
java

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


- This is a generic utility method
- Accepts a **function** (e.g., get subject or expiration) and extracts the desired claim from the token

✓ Get All Claims (Internally Used)

```
java

private Claims getAllClaimsFromToken(String token) {
    return Jwts.parser()
        .setSigningKey(secret)
        .parseClaimsJws(token)
        .getBody();
}
```

- It parses the token using the secret key
- Returns all the claims (username, issuedAt, expiration, etc.)

 This is how the server **reads** the token — but only possible with the **correct secret key**.

✓ Check if Token is Expired

```
java

private Boolean isTokenExpired(String token) {
    final Date expiration = getExpirationDateFromToken(token);
    return expiration.before(new Date());
}
```

- Gets the expiry time and compares it with the **current time**
- Returns **true** if token is **already expired**

✓ Generate Token

```
java

public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    return doGenerateToken(claims, userDetails.getUsername());
}
```

- Called during login after user is authenticated
- Generates a token with:
 - No custom claims (yet)
 - Subject = username (email or id)
 - Signed with your secret key

▼ Internally Calls:

```
java

private String doGenerateToken(Map<String, Object> claims, String subject) {
    return Jwts.builder()
        .setClaims(claims)
        .setSubject(subject)
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY * 1000))
        .signWith(SignatureAlgorithm.HS512, secret)
        .compact();
}
```

Breakdown:

Line	Meaning
<code>setClaims(...)</code>	Add custom data like roles, etc. (empty for now)
<code>setSubject(...)</code>	Add username/email as the subject of token
<code>setIssuedAt(...)</code>	Timestamp when the token was created
<code>setExpiration(...)</code>	Set token expiry to now + 5 hours
<code>signWith(...)</code>	Use secret + HS512 to sign the token
<code>compact()</code>	Convert everything to a token string (base64 encoded)

✓ Validate Token


```
java

public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = getUsernameFromToken(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
```

This checks two things:

1. ✓ Username in the token **matches the logged-in user**

2.  Token is **not expired**

If both pass → token is **valid** 

Real-World Login Example (End-to-End Flow)

1 Client sends POST /auth/login:

```
json

{
  "email": "piyush",
  "password": "piyush"
}
```

2 Backend:

- Authenticates user
- Calls `generateToken(...)` → returns:

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJwaXllc2giLCJpYXQiOiJE20TY5NDk0MDAsImV4cCI6MTY5Njk2NzQwMH0.ab
xyz
```

3 Client stores token in headers:

```
http

Authorization: Bearer <JWT_TOKEN>
```

4 For next requests:

- Backend extracts token
- Calls `validateToken(...)`
- If valid → allow access
- Else → send 401 error

Summary Table

Method	Purpose
<code>generateToken()</code>	Create JWT with username and expiration
<code>getUsernameFromToken()</code>	Extracts subject (username)
<code>getExpirationDateFromToken()</code>	Extracts expiry time
<code>isTokenExpired()</code>	Checks if token is expired
<code>validateToken()</code>	Verifies if token is valid and belongs to correct user

Method	Purpose
<code>getAllClaimsFromToken()</code>	Parses all token content using the secret

✓ Optional Add-ons You Can Try

Feature	How to Add
Add roles to claims	Put roles in the <code>claims</code> map in <code>generateToken()</code>
Refresh token	Create a method that generates a new token from old one
Add custom claims (e.g., user ID)	<code>claims.put("userId", 101)</code> before building token

Let me know if you'd like help writing the `JwtAuthenticationFilter` next — that's the class that **extracts token from request and uses this helper to validate it!**



```
package me.learning.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.MalformedJwtException;

import java.io.IOException;

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final Logger logger = LoggerFactory.getLogger(OncePerRequestFilter.class);

    @Autowired
    private JwtHelper jwtHelper;

    @Autowired
    private UserDetailsService userDetailsService;
```

```

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

    // Get the token from Authorization header
    String requestHeader = request.getHeader("Authorization");
    logger.info(" Header : {}", requestHeader);

    String username = null;
    String token = null;

    if (requestHeader != null && requestHeader.startsWith("Bearer ")) {
        token = requestHeader.substring(7); // Remove "Bearer " prefix

        try {
            username = this.jwtHelper.getUsernameFromToken(token);
        } catch (IllegalArgumentException e) {
            logger.info("Illegal Argument while fetching the username!!");
            e.printStackTrace();
        } catch (ExpiredJwtException e) {
            logger.info("Given JWT token is expired!!");
            e.printStackTrace();
        } catch (MalformedJwtException e) {
            logger.info("Invalid JWT!!");
            e.printStackTrace();
        }
    } else {
        logger.info("Invalid Header value!!");
    }

    // Once we get the token, validate it and set authentication
    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

        // Load user details
        UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

        // Validate token
        Boolean isValid = this.jwtHelper.validateToken(token, userDetails);

        if (isValid) {
            // Set authentication
            UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(userDetails, null,
                    userDetails.getAuthorities());

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

            SecurityContextHolder.getContext().setAuthentication(authentication);
        } else {

```

```
        logger.info("Validation fails!!!");
    }
}

// Continue with filter chain
filterChain.doFilter(request, response);
}
```

explain each and everything in details so that i understand the flow

Excellent, Piyush! You're now looking at the final and most crucial piece of the JWT Security puzzle: the `JwtAuthenticationFilter`.

This filter is the one that:

- ✔ Extracts the JWT from the HTTP request
- ✔ Validates it
- ✔ Authenticates the user for Spring Security

Let's go **line-by-line**, in **very simple and clear language**, so you fully understand **how a JWT-protected request is handled**.

✓ Class Declaration

```
java

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
```

- `@Component`: Makes this class a Spring-managed bean, so it can be injected into your `SecurityConfig`.
- `extends OncePerRequestFilter`:
 - This ensures that **this filter runs once per request** (not multiple times).

This is where the **JWT token from the request** is read, validated, and then used to authenticate the user.

✓ Dependencies

```
java

@Autowired
private JwtHelper jwtHelper;

@Autowired
private UserDetailsServiceImpl userDetailsService;
```

- **JwtHelper:** Used to **extract username, validate,** and **parse the token.**
- **UserDetailsService:** Loads user info (from memory or DB) based on the username extracted from token.

✓ Logger

```
java

private final Logger logger = LoggerFactory.getLogger(OncePerRequestFilter.class);
```

Used for logging messages (like token missing, token expired, etc.).

✓ Core Method: doFilterInternal(...)

This method is executed for **every incoming HTTP request**. Here's where the main JWT magic happens.

```
java

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

Let's break this into **stages**.

◆ STAGE 1: Extract Token from Header

```
java

String requestHeader = request.getHeader("Authorization");
logger.info(" Header : {}", requestHeader);
```

- Tries to read the **Authorization** header.

Example:

```
makefile

Authorization: Bearer eyJhbGciOiJIUzUxMiJ9...
```

```
java

String username = null;
String token = null;

if (requestHeader != null && requestHeader.startsWith("Bearer ")) {
    token = requestHeader.substring(7); // Remove "Bearer " prefix
```

- Removes the "Bearer " part → only keeps the raw token string.

◆ STAGE 2: Extract Username from Token

```
java

try {
    username = this.jwtHelper.getUsernameFromToken(token);
```

- Uses `JwtHelper` to extract the username (subject) from token.
- If token is expired or malformed, catches and logs errors:

```
java

} catch (IllegalArgumentException e) {
    logger.info("Illegal Argument while fetching the username!!");
    e.printStackTrace();
} catch (ExpiredJwtException e) {
    logger.info("Given JWT token is expired!!");
    e.printStackTrace();
} catch (MalformedJwtException e) {
    logger.info("Invalid JWT!!");
    e.printStackTrace();
}
```

◆ STAGE 3: Check if Spring Security is Already Authenticated

```
java

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

- If username was successfully extracted **and** no one is already authenticated:
 - Continue with authentication

◆ STAGE 4: Load User and Validate Token

```
java

UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);
Boolean isValid = this.jwtHelper.validateToken(token, userDetails);
```

- Loads user details (from in-memory or DB)
- Validates the token with:
 - Username match
 - Token not expired

◆ STAGE 5: Set Authentication in Security Context

```
java

if (isValid) {
    UsernamePasswordAuthenticationToken authentication =
```



```
new UsernamePasswordAuthenticationToken(
    userDetails, null, userDetails.getAuthorities());

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

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

- Creates an **Authentication** object (Spring's internal object representing an authenticated user)
- Sets it in the **SecurityContext**, so now:
 - ✓ Spring knows the user is authenticated
 - ✓ The request can now access secured endpoints

If validation fails:

```
java

} else {
    logger.info("Validation fails!!");
}
```

◆ STAGE 6: Continue with the Filter Chain

```
java

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

- Very important! Passes the request down the chain to the next filter (or to the controller).
- Without this, your request will hang and never reach the controller.

✓ Real Flow in Action (Example)

1 Client sends request:

```
http

GET /test
Authorization: Bearer <VALID_TOKEN>
```

2 JwtAuthenticationFilter steps:

Step	Action
1	Extracts token from header
2	Parses token to get username
3	Loads user from UserDetailsService
4	Validates token
5	Sets user as authenticated in Spring context
6	Passes request to controller

✓ Summary Table

Part	What it Does
<code>getHeader("Authorization")</code>	Extracts JWT token from request
<code>getUsernameFromToken(token)</code>	Parses JWT and extracts username
<code>loadUserByUsername()</code>	Loads user details (from DB/in-memory)
<code>validateToken()</code>	Checks if token is valid (not expired and correct user)
<code>UsernamePasswordAuthenticationToken</code>	Used to authenticate the user in Spring
<code>SecurityContextHolder.setAuthentication()</code>	Marks the user as logged in for this request
<code>filterChain.doFilter()</code>	Forwards the request to the next filter/controller

✓ What If Token is Invalid or Missing?

- If token is expired, malformed, or missing → authentication is **not set**
- So the request hits your `JwtAuthenticationEntryPoint` and gets a **401 Unauthorized**

Would you like a **visual flow diagram** of the whole request lifecycle with JWT?

Or should I help you build a Postman collection or frontend to test this securely?