Complete Guide: Building a Spring Boot JWT Authentication System

Table of Contents

Project Setup

Prerequisites

- Java 17 or higher
- Maven or Gradle
- IDE (IntelliJ IDEA, Eclipse, or VS Code)
- Postman or any API testing tool

Dependencies Required (pom.xml)

```
xml
```

```xml
<dependencies>
  <!-- Spring Boot Starter Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot Starter Security -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <!-- Spring Boot Starter Data JPA -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- H2 Database (or your preferred database) -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <!-- JWT Library -->
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
  </dependency>

  <!-- JAXB API (required for JWT) -->
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
  </dependency>
</dependencies>
```

Application Properties (application.properties)

```
properties
```

```
# Server Configuration
server.port=8080

# Database Configuration (H2 example)
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA Configuration
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# H2 Console (optional, for testing)
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Implementation Guide

Step 1: Create the User Entity

File: `com/pone/jwtauth/Entity/User.java`

This is your database model representing users.

```
java
```

```java
package com.pone.jwtauth.Entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(generator = "increment")
    private Long id;
    private String username;
    private String password;

    // Getters and Setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Key Points:

- `@Entity` marks this as a JPA entity
- `@Table(name = "users")` specifies the database table name
- `@Id` marks the primary key
- `@GeneratedValue` auto-generates ID values

Step 2: Create DTOs (Data Transfer Objects)

DTOs are used to transfer data between client and server, separating the API layer from the database layer.

2.1 RegisterRequestDTO

File: `com/pone/jwtauth/DTO/RegisterRequestDTO.java`

```java
package com.pone.jwtauth.DTO;

public class RegisterRequestDTO {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

2.2 RegisterResponseDTO

File: `com/pone/jwtauth/DTO/RegisterResponseDTO.java`

```java
```

```java
package com.pone.jwtauth.DTO;

public class RegisterResponseDTO {
    private Long id;
    private String username;
    private String message;

    public RegisterResponseDTO(Long id, String username, String message) {
        this.id = id;
        this.username = username;
        this.message = message;
    }
    public RegisterResponseDTO() {
    }

    // Getters and Setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}
```

2.3 LoginRequestDTO

File: `com/pone/jwtauth/DTO/LoginRequestDTO.java`

```
java
```

```java
package com.pone.jwtauth.DTO;

public class LoginRequestDTO {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

2.4 AuthResponseDTO

File: `com/pone/jwtauth/DTO/AuthResponseDTO.java`

java

```java
package com.pone.jwtauth.DTO;

public class AuthResponseDTO {
    private String token;

    public AuthResponseDTO(String token) {
        this.token = token;
    }
    public AuthResponseDTO() {
    }

    public String getToken() {
        return token;
    }
    public void setToken(String token) {
        this.token = token;
    }
}
```

Step 3: Create Mappers

Mappers convert between DTOs and Entities.

3.1 toEntity Mapper

File: `com/pone/jwtauth/Mapper/toEntity.java`

```java
package com.pone.jwtauth.Mapper;

import org.springframework.security.crypto.password.PasswordEncoder;
import com.pone.jwtauth.Entity.User;
import com.pone.jwtauth.DTO.RegisterRequestDTO;

public class toEntity {
    public static User mapRegisterRequestDTOtoUser(RegisterRequestDTO dto) {
        User user = new User();
        // Create password encoder for encrypting password
        PasswordEncoder passwordEncoder = new org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder();

        user.setUsername(dto.getUsername());
        // IMPORTANT: Encrypt password before storing
        user.setPassword(passwordEncoder.encode(dto.getPassword()));
        return user;
    }
}
```

Key Point: Password is encrypted using BCryptPasswordEncoder before saving to database.

3.2 toResponse Mapper

File: `com/pone/jwtauth/Mapper/toResponse.java`

```java
```

```
package com.pone.jwtauth.Mapper;

import com.pone.jwtauth.DTO.AuthResponseDTO;
import com.pone.jwtauth.DTO.RegisterResponseDTO;
import com.pone.jwtauth.Entity.User;

public class toResponse {
    // Map User entity to RegisterResponseDTO
    public static RegisterResponseDTO mapToRegisterResponseDTO(User user, String message) {
        RegisterResponseDTO responseDTO = new RegisterResponseDTO();
        responseDTO.setId(user.getId());
        responseDTO.setUsername(user.getUsername());
        responseDTO.setMessage(message);
        return responseDTO;
    }

    // Map JWT token to AuthResponseDTO
    public static AuthResponseDTO mapToAuthResponseDTO(String token) {
        AuthResponseDTO authResponseDTO = new AuthResponseDTO();
        authResponseDTO.setToken(token);
        return authResponseDTO;
    }
}
```

3.3 UtilDTO (Helper)

File: `com/pone/jwtauth/util/dtoutil/UtilDTO.java`

```java
package com.pone.jwtauth.util.dtoutil;

import com.pone.jwtauth.DTO.LoginRequestDTO;

public class UtilDTO {
    public static String toUsername(LoginRequestDTO dto) {
        return dto.getUsername();
    }
    public static String toPassword(LoginRequestDTO dto) {
        return dto.getPassword();
    }
}
```

Step 4: Create Repository

File: `com/pone/jwtauth/Repo/UserRepo.java`

```java
package com.pone.jwtauth.Repo;

import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import com.pone.jwtauth.Entity.User;

public interface UserRepo extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
}
```

Key Point: JpaRepository provides CRUD operations. The `findByUsername` method is used for authentication.

Step 5: Create UserService

File: `com/pone/jwtauth/Service/UserService.java`

This is a crucial service that handles user operations and implements Spring Security's `UserDetailsService`.

```java
```

```java
package com.pone.jwtauth.Service;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.pone.jwtauth.DTO.RegisterRequestDTO;
import com.pone.jwtauth.DTO.RegisterResponseDTO;
import com.pone.jwtauth.Entity.User;
import com.pone.jwtauth.Mapper.toEntity;
import com.pone.jwtauth.Mapper.toResponse;
import com.pone.jwtauth.Repo.UserRepo;

@Service
public class UserService implements UserDetailsService {
    private final UserRepo userRepo;

    public UserService(UserRepo userRepo) {
        this.userRepo = userRepo;
    }

    // Register new user
    public RegisterResponseDTO registerUser(RegisterRequestDTO dto) {
        // Check if user already exists
        if(userRepo.findByUsername(dto.getUsername()).isPresent()) {
            System.out.println("Username already exists: " + dto.getUsername());
            return new RegisterResponseDTO(null, null, "Username already exists");
        }

        // Map DTO to entity (password gets encrypted in mapper)
        User user = toEntity.mapRegisterRequestDTOtoUser(dto);

        // Save user to database
        userRepo.save(user);

        // Return response DTO
        return toResponse.mapToRegisterResponseDTO(user, "User registered successfully");
    }

    // Load user by username - CRUCIAL for Spring Security authentication
    @Override
    public UserDetails loadUserByUsername(String usernameOrEmail)
            throws UsernameNotFoundException {
        // Fetch user from database
        User user = userRepo.findByUsername(usernameOrEmail)
```

```
            .orElseThrow(() -> new UsernameNotFoundException(
                "User not found with username: " + usernameOrEmail
            ));

        // Return Spring Security's UserDetails object
        return new org.springframework.security.core.userdetails.User(
            user.getUsername(),
            user.getPassword(),
            new java.util.ArrayList<>() // Empty authorities list
        );
    }

    // Helper methods
    public boolean userExists(String username) {
        return userRepo.findByUsername(username).isPresent();
    }

    public User getUserEntity(String username) {
        return userRepo.findByUsername(username)
            .orElseThrow(() -> new RuntimeException("User not found"));
    }

    public User saveUserDirect(User user) {
        return userRepo.save(user);
    }
}
```

How loadUserByUsername works:

1. Spring Security's AuthenticationManager calls this method during login
2. It fetches the user from the database
3. Returns a UserDetails object containing username, encrypted password, and authorities
4. Spring Security compares the provided password with the encrypted password

---

Step 6: Create JwtUtil

File: `com/pone/jwtauth/util/JwtUtil.java`

This utility class handles all JWT operations - generating, validating, and extracting information from tokens.

```
java
```

```java
package com.pone.jwtauth.util;

import java.util.Date;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JwtUtil {
  // Secret key for signing tokens (should be stored securely in production)
  private final String SECRET = "hsrnjgnsongjajobgjbajbguibuirbiwbeihbtibitwibeuitwuinjfgjnjgdfgbfdt";

  // Token expiration time: 86400000 ms = 24 hours
  private final long EXPIRATION = 86400000;

  // Generate JWT token for a user
  public String generateToken(String username) {
    return Jwts.builder()
        .setSubject(username)                   // Set username as subject
        .setIssuedAt(new Date())                // Set issue time
        .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION)) // Set expiration
        .signWith(SignatureAlgorithm.HS256, SECRET)    // Sign with algorithm and secret
        .compact();                             // Build the token
  }

  // Extract username from token
  public String extractUsername(String token) {
    return getClaims(token).getSubject();
  }

  // Validate token
  public boolean validateToken(String token) {
    try {
      getClaims(token);
      return true;
    } catch (JwtException | IllegalArgumentException e) {
      return false;
    }
  }

  // Parse and return claims from token
  private Claims getClaims(String token) {
    return Jwts.parser()
        .setSigningKey(SECRET)
```

```
                .parseClaimsJws(token)
                .getBody();
    }
}
```

Key Points:

- `generateToken()`: Creates a new JWT with username, issue date, and expiration
- `extractUsername()`: Retrieves username from token claims
- `validateToken()`: Checks if token is valid and not expired
- `getClaims()`: Internal method to parse token and extract claims

Step 7: Create JwtAuthFilter

File: `com/pone/jwtauth/Filter/JwtAuthFilter.java`

This filter intercepts every HTTP request to validate JWT tokens.

```
java
```

```java
package com.pone.jwtauth.Filter;

import java.io.IOException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import com.pone.jwtauth.Service.UserService;
import com.pone.jwtauth.util.JwtUtil;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class JwtAuthFilter extends OncePerRequestFilter {
    private final UserService userService;
    private final JwtUtil jwtUtil;

    public JwtAuthFilter(UserService userService, JwtUtil jwtUtil) {
        this.userService = userService;
        this.jwtUtil = jwtUtil;
    }

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

        // Extract Authorization header
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;

        // Check if header exists and starts with "Bearer "
        if(authHeader != null && authHeader.startsWith("Bearer ")) {
            // Extract token (remove "Bearer " prefix)
            token = authHeader.substring(7);

            try {
                // Extract username from token
                username = jwtUtil.extractUsername(token);
                System.out.println("Authenticated user: " + username);
                // Load user details
                UserDetails userDetails = userService.loadUserByUsername(username);

                // Create authentication token
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(
                        userDetails,
                        null,
                        userDetails.getAuthorities()
                    );

                // Set authentication in security context
                SecurityContextHolder.getContext().setAuthentication(authentication);

            } catch(Exception e) {
                System.out.println("Invalid token: " + e.getMessage());
            }
        }

        // Continue with the filter chain (like next() in Express.js)
        filterChain.doFilter(request, response);
    }
}
```

Flow:

1. Intercepts every request
2. Checks for "Authorization: Bearer <token>" header
3. Extracts and validates token
4. Loads user details from database
5. Sets authentication in Spring Security context
6. Passes request to next filter/controller

Step 8: Create SecurityConfig

File: `com/pone/jwtauth/Config/SecurityConfig.java`

This configures Spring Security for JWT authentication.

```java
```

```java
package com.pone.jwtauth.Config;

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.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import com.pone.jwtauth.Filter.JwtAuthFilter;
import com.pone.jwtauth.Service.UserService;
import com.pone.jwtauth.util.JwtUtil;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
  private final JwtAuthFilter jwtAuthFilter;

  public SecurityConfig(UserService userService, JwtUtil jwtUtil) {
    this.jwtAuthFilter = new JwtAuthFilter(userService, jwtUtil);
  }

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
      // Disable CSRF (not needed for stateless JWT APIs)
      .csrf(csrf -> csrf.disable())

      // Set session management to stateless
      .sessionManagement(session ->
        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
      )

      // Configure authorization rules
      .authorizeHttpRequests(auth -> auth
        .requestMatchers("/auth/**").permitAll()  // Allow /auth endpoints
        .anyRequest().authenticated()             // Require auth for all others
      )

      // Add JWT filter before UsernamePasswordAuthenticationFilter
      .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
```

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

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

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

Key Configurations:

- **CSRF disabled**: Not needed for stateless JWT authentication
- **Stateless sessions**: No server-side sessions
- **Public endpoints**: `/auth/signup` and `/auth/login` are accessible without authentication
- **Protected endpoints**: All other endpoints require valid JWT
- **Filter order**: JwtAuthFilter runs before standard authentication filter

---

Step 9: Create AuthService

File: `com/pone/jwtauth/Service/AuthService.java`

This service handles the login authentication logic.

java

```java
package com.pone.jwtauth.Service;

import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Service;
import com.pone.jwtauth.DTO.LoginRequestDTO;
import com.pone.jwtauth.util.dtoutil.UtilDTO;

@Service
public class AuthService {
    private final AuthenticationManager authenticationManager;

    public AuthService(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    public String login(LoginRequestDTO loginDTO) {
        // Extract credentials from DTO
        String username = UtilDTO.toUsername(loginDTO);
        String password = UtilDTO.toPassword(loginDTO);

        // Create authentication token with credentials
        // AuthenticationManager will:
        // 1. Call UserService.loadUserByUsername()
        // 2. Compare provided password with stored encrypted password
        // 3. Return authenticated object if valid
        Authentication auth = authenticationManager.authenticate(
            new org.springframework.security.authentication.UsernamePasswordAuthenticationToken(
                username,
                password
            )
        );

        System.out.println("Authentication successful for user: " + auth.getName());

        // Validate that authenticated username matches request
        if (!auth.getName().equals(loginDTO.getUsername())) {
            throw new RuntimeException("Invalid credentials");
        }

        return loginDTO.getUsername();
    }
}
```

Authentication Flow:

1. Extract username and password from DTO
2. Create `UsernamePasswordAuthenticationToken`
3. `AuthenticationManager` validates credentials by:
    - Calling `UserService.loadUserByUsername()`
    - Using `PasswordEncoder` to compare passwords
4. Returns authenticated username if successful
5. Throws exception if authentication fails

Step 10: Create AuthController

File: `com/pone/jwtauth/Controller/AuthController.java`

This REST controller exposes the authentication endpoints.

```
java
```

```java
package com.pone.jwtauth.Controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import com.pone.jwtauth.DTO.AuthResponseDTO;
import com.pone.jwtauth.DTO.LoginRequestDTO;
import com.pone.jwtauth.DTO.RegisterRequestDTO;
import com.pone.jwtauth.DTO.RegisterResponseDTO;
import com.pone.jwtauth.Mapper.toResponse;
import com.pone.jwtauth.Service.AuthService;
import com.pone.jwtauth.Service.UserService;
import com.pone.jwtauth.util.JwtUtil;

@RestController
@RequestMapping("/auth")
public class AuthController {
    private final AuthService authService;
    private final UserService userService;
    private final JwtUtil jwtUtil;

    public AuthController(AuthService authService, UserService userService, JwtUtil jwtUtil) {
        this.authService = authService;
        this.userService = userService;
        this.jwtUtil = jwtUtil;
    }

    // Endpoint: POST /auth/signup
    @PostMapping("/signup")
    public RegisterResponseDTO registerUser(@RequestBody RegisterRequestDTO registerDTO) {
        return userService.registerUser(registerDTO);
    }

    // Endpoint: POST /auth/login
    @PostMapping("/login")
    public AuthResponseDTO login(@RequestBody LoginRequestDTO request) {
        // Validate credentials and get username
        String validatedUsername = authService.login(request);

        // Generate JWT token
        String token = jwtUtil.generateToken(validatedUsername);

        // Return token in response DTO
        return toResponse.mapToAuthResponseDTO(token);
    }
}
```

Request-Response Flow

1. Registration Flow

```
Client Request:
POST /auth/signup
Content-Type: application/json
{
  "username": "john",
  "password": "secret123"
}

Flow:
1. AuthController.registerUser() receives RegisterRequestDTO
   ↓
2. UserService.registerUser() is called
   ↓
3. Check if username exists in database
   - If exists: return error message
   - If not: continue
   ↓
4. toEntity.mapRegisterRequestDTOtoUser()
   - Creates User entity
   - Encrypts password with BCryptPasswordEncoder
   ↓
5. UserRepo.save() saves user to database
   ↓
6. toResponse.mapToRegisterResponseDTO() creates response
   ↓
7. Return RegisterResponseDTO

Server Response:
{
  "id": 1,
  "username": "john",
  "message": "User registered successfully"
}
```

---

2. Login Flow

```
Client Request:
POST /auth/login
Content-Type: application/json
{
  "username": "john",
  "password": "secret123"
}

Flow:
1. AuthController.login() receives LoginRequestDTO
   ↓
2. AuthService.login() is called
   ↓
3. Extract username and password
   ↓
4. Create UsernamePasswordAuthenticationToken(username, password)
   ↓
5. AuthenticationManager.authenticate() is called
   ↓
6. Internally calls UserService.loadUserByUsername()
   - Fetches user from database
   - Returns UserDetails with encrypted password
   ↓
7. PasswordEncoder compares provided password with stored password
   ↓
8. If valid: authentication succeeds
   If invalid: throws BadCredentialsException
   ↓
9. Return validated username to AuthController
   ↓
10. JwtUtil.generateToken(username) creates JWT token
   ↓
11. toResponse.mapToAuthResponseDTO() wraps token
   ↓
12. Return AuthResponseDTO

Server Response:
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJqb2huIiwiaWF0IjoxNzA..."
}
```

3. Accessing Protected Endpoint

```
Client Request:
GET /api/protected-resource
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Flow:
1. Request intercepted by JwtAuthFilter.doFilterInternal()
   ↓
2. Extract "Authorization" header
   ↓
3. Check if header starts with "Bearer "
   ↓
4. Extract token (remove "Bearer " prefix)
   ↓
5. JwtUtil.extractUsername(token)
   - Parses token
   - Returns username from claims
   ↓
6. JwtUtil.validateToken(token)
   - Validates signature
   - Checks expiration
   ↓
7. UserService.loadUserByUsername(username)
   - Loads user details from database
   ↓
8. Create UsernamePasswordAuthenticationToken
   ↓
9. Set authentication in SecurityContextHolder
   ↓
10. filterChain.doFilter() passes request to controller
   ↓
11. Controller processes request with authenticated context
   ↓
12. Returns response to client

If token is invalid:
- Authentication is NOT set
- Request reaches controller
- Spring Security blocks access (401 Unauthorized)
```

Testing the Application

1. Start the Application

Run the main application class:

```java
package com.pone.jwtauth;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class JwtauthApplication {
    public static void main(String[] args) {
        SpringApplication.run(JwtauthApplication.class, args);
    }
}
```

The application starts on `http://localhost:8080`

---

2. Test Registration

Using Postman or cURL:

```bash
POST http://localhost:8080/auth/signup
Content-Type: application/json

{
  "username": "testuser",
  "password": "password123"
}
```

Expected Response:

```json
{
  "id": 1,
  "username": "testuser",
  "message": "User registered successfully"
}
```

---

3. Test Login

```bash
POST http://localhost:8080/auth/login
Content-Type: application/json

{
  "username": "testuser",
  "password": "password123"
}
```

Expected Response:

```json
{
  "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0dXNlciIsImlhdCI6MTcwNjE..."
}
```

Copy this token for the next step.

---

4. Test Protected Endpoint

First, create a test controller:

```java
package com.pone.jwtauth.Controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class TestController {

    @GetMapping("/protected")
    public String protectedEndpoint() {
        return "This is a protected endpoint. You are authenticated!";
    }
}
```

Test with token:

```bash
GET http://localhost:8080/api/protected
Authorization: Bearer <paste-your-token-here>
```

Expected Response:

```
This is a protected endpoint. You are authenticated!
```

Test without token:

```bash
GET http://localhost:8080/api/protected
```

Expected Response:

```
401 Unauthorized
```

Project Structure

```
src/main/java/com/pone/jwtauth/
├── JwtauthApplication.java        # Main application entry point
├── Config/
│   └── SecurityConfig.java        # Spring Security configuration
├── Controller/
│   ├── AuthController.java        # Authentication endpoints
│   └── TestController.java        # Test protected endpoint
├── DTO/
│   ├── RegisterRequestDTO.java      # Registration request
│   ├── RegisterResponseDTO.java     # Registration response
│   ├── LoginRequestDTO.java        # Login request
│   └── AuthResponseDTO.java        # Login response (token)
├── Entity/
│
```