JWT Authentication System - Complete Code Documentation

Table of Contents

Project Overview

Project Name: Jwt_Blog
Technology Stack: Spring Boot, Spring Security, JWT, JPA/Hibernate
Authentication Method: JWT Token-based Authentication
Database: MySQL (or any JPA-compatible database)

Key Features

- User registration with password encryption
- JWT token generation on login
- Token-based authentication for protected endpoints
- Stateless session management
- RESTful API design

Complete Code Listing

1. JwtAuthFilter.java

Package: `com.blogjwt.Jwt_Blog.Config`
Purpose: Custom filter to intercept requests and validate JWT tokens

```
java
```

```java
package com.blogjwt.Jwt_Blog.Config;

import com.blogjwt.Jwt_Blog.Service.UserService;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
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;

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

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

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

        String header = request.getHeader("Authorization");

        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);

            if (jwtUtil.validateToken(token)) {
                String username = jwtUtil.extractUsername(token);
                System.out.println("validate token username" + username);

                UserDetails userDetails =
                        userService.loadUserByUsername(username);

                UsernamePasswordAuthenticationToken authentication =
                        new UsernamePasswordAuthenticationToken(
                                userDetails,
                                null,
                                userDetails.getAuthorities()
```

```
                    );

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

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

Key Components:

- Extends `OncePerRequestFilter` - ensures filter runs once per request
- Extracts JWT token from Authorization header
- Validates token and loads user details
- Sets authentication in SecurityContext

---

2. JwtUtil.java

Package: `com.blogjwt.Jwt_Blog.Config`
Purpose: Utility class for JWT token operations

```java
```

```java
package com.blogjwt.Jwt_Blog.Config;

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

import java.util.Date;

@Component
public class JwtUtil {

    private final String SECRET = "hsrnjgnsongjajobgjbajbguibuirbiwbeihbtibitwibeuitwuinjfgjnjgdfgbfdt";
    private final long EXPIRATION = 86400000; // 1 day

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION))
                .signWith(SignatureAlgorithm.HS256, SECRET)
                .compact();
    }

    public String extractUsername(String token) {
        return getClaims(token).getSubject();
    }

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

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

Key Components:

- **SECRET**: Secret key for signing tokens (should be in environment variables)
- **EXPIRATION**: Token validity period (24 hours)
- **generateToken()**: Creates JWT with username, issue date, and expiration
- **validateToken()**: Verifies token signature and validity
- **extractUsername()**: Parses token to get username

---

3. SecurityConfig.java

Package: `com.blogjwt.Jwt_Blog.Config`
Purpose: Spring Security configuration

```
java
```

```
package com.blogjwt.Jwt_Blog.Config;

import com.blogjwt.Jwt_Blog.Service.UserService;
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;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final JwtAuthFilter jwtAuthFilter;
    private final UserService userService;

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

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

        http
                // Disable CSRF for JWT-based APIs
                .csrf(csrf -> csrf.disable())

                // Configure stateless session
                .sessionManagement(session ->
                        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                )

                // Authorization rules
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/auth/**").permitAll()
                        .anyRequest().authenticated()
                )

                // Add JWT filter before UsernamePasswordAuthenticationFilter
```

```
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

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

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

Key Components:

- **CSRF disabled**: Not needed for stateless JWT authentication
- **Stateless sessions**: No server-side session storage
- **Public endpoints**: `/auth/**` accessible without authentication
- **JwtAuthFilter**: Added before standard authentication filter
- **BCrypt**: Password hashing algorithm

---

4. AuthController.java

Package: `com.blogjwt.Jwt_Blog.Controller`
Purpose: Handles user registration and login

```java
```

```java
package com.blogjwt.Jwt_Blog.Controller;

import com.blogjwt.Jwt_Blog.Config.JwtUtil;
import com.blogjwt.Jwt_Blog.DTO.LoginRequestDTO;
import com.blogjwt.Jwt_Blog.DTO.LoginResponseDTO;
import com.blogjwt.Jwt_Blog.DTO.SignUpDTO;
import com.blogjwt.Jwt_Blog.Entity.User;
import com.blogjwt.Jwt_Blog.Mapper.AuthMapper;
import com.blogjwt.Jwt_Blog.Service.UserService;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.*;

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

    private final AuthenticationManager authenticationManager;
    private final JwtUtil jwtUtil;
    private final UserService userService;
    private final PasswordEncoder passwordEncoder;

    public AuthController(AuthenticationManager authenticationManager, JwtUtil jwtUtil,
                          UserService userService, PasswordEncoder passwordEncoder) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
        this.userService = userService;
        this.passwordEncoder = passwordEncoder;
    }

    @PostMapping("/signup")
    public String signup(@RequestBody SignUpDTO request) {
        if (userService.userExists(request.getUsername())) {
            return "Username already exists";
        }
        User user = new User();
        user.setUsername(request.getUsername());
        user.setEmailId(request.getEmailId());
        user.setPassword(passwordEncoder.encode(request.getPassword()));

        User catchCreatedUser = userService.saveUser(user);

        return "User registered successfully";
    }

    @PostMapping("/login")
    public LoginResponseDTO login(@RequestBody LoginRequestDTO request) {
        try {
            String username = AuthMapper.toUsername(request);
            String pass = AuthMapper.toPassword(request);
            Authentication auth = authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(username, pass)
            );

            System.out.println(auth.getName());
            if (!auth.getName().equals(request.getUsername())) {
                throw new RuntimeException("Invalid credentials");
            }
            String token = jwtUtil.generateToken(request.getUsername());
            return AuthMapper.toReponse(token);

        } catch (AuthenticationException ex) {
            throw new RuntimeException("Invalid username or password");
        }
    }
}
```

Key Components:

- **POST /auth/signup**: Registers new user with encrypted password
- **POST /auth/login**: Authenticates user and returns JWT token
- Uses AuthenticationManager to verify credentials
- Returns JWT token on successful login

---

5. DTOs (Data Transfer Objects)

LoginRequestDTO.java

```
java
```

```java
package com.blogjwt.Jwt_Blog.DTO;

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

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return this.username;
    }

    public String getPassword() {
        return this.password;
    }

    public String getEmailId() {
        return emailId;
    }

    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }
}
```

LoginResponseDTO.java

java

```
package com.blogjwt.Jwt_Blog.DTO;

public class LoginResponseDTO {
    private final String token;

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

    public String getToken() {
        return token;
    }
}
```

SignUpDTO.java

```
java
```

```java
package com.blogjwt.Jwt_Blog.DTO;

public class SignUpDTO {
    private String username;
    private String emailId;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getEmailId() {
        return emailId;
    }

    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

---

6. User Entity

Package: `com.blogjwt.Jwt_Blog.Entity`
Purpose: Database entity for users

```
java
```

```java
package com.blogjwt.Jwt_Blog.Entity;

import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(unique = true, nullable = false)
    private String emailId;

    @Column(nullable = false)
    private String password;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Blog> blogs = new ArrayList<>();

    // 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 getEmailId() {
        return emailId;
    }
    public void setEmailId(String emailId) {
        this.emailId = emailId;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public List<Blog> getBlogs() {
        return blogs;
    }

    public void setBlogs(List<Blog> blogs) {
        this.blogs = blogs;
    }
}
```

Database Schema:

- **id**: Auto-generated primary key
- **username**: Unique, required
- **emailId**: Unique, required
- **password**: BCrypt hashed password
- **blogs**: One-to-many relationship with Blog entity

---

7. AuthMapper.java

Package: `com.blogjwt.Jwt_Blog.Mapper`
Purpose: Maps between DTOs

```java
```

```
package com.blogjwt.Jwt_Blog.Mapper;

import com.blogjwt.Jwt_Blog.DTO.LoginRequestDTO;
import com.blogjwt.Jwt_Blog.DTO.LoginResponseDTO;

public class AuthMapper {
    public static String toUsername(LoginRequestDTO dto) {
        return dto.getUsername();
    }

    public static String toPassword(LoginRequestDTO dto) {
        return dto.getPassword();
    }

    public static LoginResponseDTO toReponse(String token) {
        return new LoginResponseDTO(token);
    }
}
```

---

8. UserService.java

Package: `com.blogjwt.Jwt_Blog.Service`
Purpose: User business logic and UserDetailsService implementation

```
java
```

```java
package com.blogjwt.Jwt_Blog.Service;

import com.blogjwt.Jwt_Blog.Entity.User;
import com.blogjwt.Jwt_Blog.Repository.UserRepository;
import org.jspecify.annotations.NonNull;
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 java.util.Collections;

@Service
public class UserService implements UserDetailsService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(@NonNull String username)
            throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username)
                .orElseThrow(() ->
                        new UsernameNotFoundException("User not found"));

        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                Collections.emptyList()
        );
    }

    public boolean userExists(String username) {
        return userRepository.findByUsername(username).isPresent();
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }

    public User getUserEntity(String username) {
        return userRepository.findByUsername(username).orElseThrow();
    }
}
```

Key Methods:

- **loadUserByUsername()**: Required by Spring Security for authentication
- **userExists()**: Checks if username is already taken
- **saveUser()**: Persists user to database
- **getUserEntity()**: Retrieves user entity by username

Code Explanation

How Authentication Works

1. User Registration Flow

```
Client sends POST /auth/signup
   ↓
AuthController receives SignUpDTO
   ↓
Check if username exists
   ↓
Create User entity
   ↓
Hash password with BCrypt
   ↓
Save to database via UserService
   ↓
Return success message
```

Code Flow:

```
java
```

```
// Step 1: Check existence
if (userService.userExists(request.getUsername())) {
    return "Username already exists";
}

// Step 2: Create user
User user = new User();
user.setUsername(request.getUsername());
user.setEmailId(request.getEmailId());

// Step 3: Hash password
user.setPassword(passwordEncoder.encode(request.getPassword()));

// Step 4: Save
userService.saveUser(user);
```

---

2. User Login Flow

```
Client sends POST /auth/login
    ↓
AuthController receives LoginRequestDTO
    ↓
AuthenticationManager authenticates
    ↓
UserService loads user from database
    ↓
Password verified (BCrypt compare)
    ↓
Generate JWT token with JwtUtil
    ↓
Return token in LoginResponseDTO
```

Code Flow:

```java
```

```
// Step 1: Create authentication token
Authentication auth = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(username, password)
);

// Step 2: AuthenticationManager internally:
// - Calls UserService.loadUserByUsername()
// - Compares passwords using BCrypt
// - Returns Authentication object if successful

// Step 3: Generate JWT
String token = jwtUtil.generateToken(request.getUsername());

// Step 4: Return response
return new LoginResponseDTO(token);
```

---

3. Protected Request Flow

```
Client sends GET /api/protected
Header: Authorization: Bearer <token>
    ↓
JwtAuthFilter intercepts request
    ↓
Extract token from header
    ↓
Validate token with JwtUtil
    ↓
Extract username from token
    ↓
Load UserDetails from UserService
    ↓
Create Authentication object
    ↓
Set in SecurityContext
    ↓
Continue to controller
```

Code Flow:

```
java
```

```
// Step 1: Extract token
String header = request.getHeader("Authorization");
String token = header.substring(7); // Remove "Bearer "

// Step 2: Validate
if (jwtUtil.validateToken(token)) {
    // Step 3: Extract username
    String username = jwtUtil.extractUsername(token);

    // Step 4: Load user
    UserDetails userDetails = userService.loadUserByUsername(username);

    // Step 5: Create authentication
    UsernamePasswordAuthenticationToken authentication =
        new UsernamePasswordAuthenticationToken(
            userDetails, null, userDetails.getAuthorities()
        );

    // Step 6: Set in context
    SecurityContextHolder.getContext().setAuthentication(authentication);
}
```

JWT Token Structure

Generated Token Example:

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2huZG9lIiwiaWF0IjoxNzA2MTIzNDU2LCJleHAiOjE3MDYyMDk4NTZ9.signature
```

Decoded:

Header:

```json
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload:

```json
```

```
{
  "sub": "johndoe",
  "iat": 1706123456,
  "exp": 1706209856
}
```

Signature:

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  SECRET
)
```

Security Context Flow

```java
// After successful JWT validation:
SecurityContextHolder.getContext().setAuthentication(authentication);

// Later in controllers:
@GetMapping("/profile")
public String getProfile() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();
    String username = auth.getName(); // Returns authenticated username
    return "Profile of " + username;
}
```

How It Works

Complete Request Lifecycle

Scenario 1: New User Registration

Request:

```http
```

```
POST /auth/signup HTTP/1.1
Content-Type: application/json

{
  "username": "johndoe",
  "emailId": "john@example.com",
  "password": "mypassword123"
}
```

Processing:

1. SecurityFilterChain allows `/auth/**` without authentication
2. AuthController.signup() receives request
3. Checks if "johndoe" exists in database
4. Creates new User entity
5. Hashes "mypassword123" with BCrypt → `$2a$10$...`
6. Saves to database

Response:

```
User registered successfully
```

Database State:

```
users table:
id | username | emailId          | password (BCrypt hash)
1  | johndoe  | john@example.com | $2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
```

Scenario 2: User Login

Request:

```http
POST /auth/login HTTP/1.1
Content-Type: application/json

{
  "username": "johndoe",
  "password": "mypassword123"
}
```

Processing:

1. AuthController.login() receives request
2. Creates UsernamePasswordAuthenticationToken
3. AuthenticationManager.authenticate() is called
4. Internally calls UserService.loadUserByUsername("johndoe")
5. UserRepository.findByUsername() queries database
6. Returns User entity
7. BCrypt compares "mypassword123" with stored hash
8. If match: Authentication successful
9. JwtUtil.generateToken("johndoe") creates JWT
10. Token includes: subject=johndoe, iat=now, exp=now+24hours
11. Signs with SECRET key

Response:

```json
{
  "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2huZG9lIiwiaWF0IjoxNzA2MTIzNDU2LCJleHAiOjE3MDYyMDk4NTZ
}
```

Scenario 3: Access Protected Resource

Request:

```http
GET /api/blogs HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2huZG9lIiwiaWF0IjoxNzA2MTIzNDU2LCJleHAiOjE3MDYy
```

Processing:

1. Request enters Spring Security Filter Chain
2. JwtAuthFilter.doFilterInternal() executes
3. Extracts "Authorization" header
4. Removes "Bearer " prefix → gets token
5. JwtUtil.validateToken() checks:
   - Valid signature (using SECRET)

    - Not expired
    - Proper format
6. JwtUtil.extractUsername() → "johndoe"
7. UserService.loadUserByUsername("johndoe")
8. Creates Authentication object with UserDetails
9. Sets in SecurityContextHolder
10. Request continues to controller
11. Controller can access authenticated user

Response:

```json
{
  "blogs": [...]
}
```

Password Hashing Deep Dive

Registration:

```java
String plainPassword = "mypassword123";
String hashedPassword = passwordEncoder.encode(plainPassword);
// Result: $2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
```

BCrypt Format:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
│  │ │  │                             │
│  │ │  └─ Salt (22 chars)                 │
│  │ └────── Work factor (2^10 = 1024 iterations)   │
│  └───────── BCrypt version                  │
└────────────── Algorithm identifier              │
                                        │
                          Hash (31 chars)┘
```

Login Verification:

```java
```

```
String inputPassword = "mypassword123";
String storedHash = "$2a$10$N9qo8uLOick...";

boolean matches = passwordEncoder.matches(inputPassword, storedHash);
// BCrypt hashes inputPassword with the salt from storedHash
// Compares the results
// Returns true if they match
```

Testing Guide

Prerequisites

- Java 17+
- MySQL running
- Postman or cURL

Database Setup

```sql
CREATE DATABASE blog_db;

USE blog_db;

-- Table is created automatically by JPA
-- But you can verify with:
DESCRIBE users;
```

Test Cases

Test 1: User Registration

Request:

```bash
curl -X POST http://localhost:8080/auth/signup \
  -H "Content-Type: application/json" \
  -d '{
    "username": "testuser",
    "emailId": "test@example.com",
    "password": "password123"
  }'
```

Expected Response:

```
User registered successfully
```

Verify in Database:

```sql
SELECT * FROM users WHERE username = 'testuser';
```

Test 2: Duplicate Username

Request:

```bash
curl -X POST http://localhost:8080/auth/signup \
  -H "Content-Type: application/json" \
  -d '{
    "username": "testuser",
    "emailId": "another@example.com",
    "password": "password456"
  }'
```

Expected Response:

```
Username already exists
```

Test 3: Successful Login

Request:

```bash
curl -X POST http://localhost:8080/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "testuser",
    "password": "password123"
  }'
```

Expected Response:

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

Save this token for next tests

Test 4: Invalid Login

Request:

```bash
curl -X POST http://localhost:8080/auth/login \
  -H "Content-Type: application/json" \
  -d '{
    "username": "testuser",
    "password": "wrongpassword"
  }'
```

Expected Response:

```
500 Internal Server Error
Invalid username or password
```

Test 5: Access Protected Endpoint (With Token)

Request:

```bash
curl -X GET http://localhost:8080/api/blogs \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..."
```

Expected Response:

```
200 OK
(Blog data or empty list)
```

Test 6: Access Protected Endpoint (Without Token)

Request:

```bash
curl -X GET http://localhost:8080/api/blogs
```

Expected Response:

```
403 Forbidden
```

Postman Collection

Import this JSON:

```json
```

```json
{
 "info": {
   "name": "JWT Auth API",
   "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
 },
 "item": [
   {
    "name": "Signup",
    "request": {
      "method": "POST",
      "header": [],
      "body": {
        "mode": "raw",
        "raw": "{\n  \"username\": \"testuser\",\n  \"emailId\": \"test@example.com\",\n  \"password\": \"password123\"\n}",
        "options": {
          "raw": {
            "language": "json"
          }
        }
      },
      "url": {
        "raw": "http://localhost:8080/auth/signup",
        "protocol": "http",
        "host": ["localhost"],
        "port": "8080",
        "path": ["auth", "signup"]
      }
    }
   },
   {
    "name": "Login",
    "request": {
      "method": "POST",
      "header": [],
      "body": {
        "mode": "raw",
        "raw": "{\n  \"username\": \"testuser\",\n  \"password\": \"password123\"\n}",
        "options": {
          "raw": {
            "language": "json"
          }
        }
      },
      "url": {
        "raw": "http://localhost:8080/auth/login",
        "protocol": "http",
        "host": ["localhost"],
        "port": "8080",
        "path": ["auth", "login"]
      }
    }
  },
  {
    "name": "Get Blogs (Protected)",
    "request": {
      "method": "GET",
      "header": [
        {
          "key": "Authorization",
          "value": "Bearer {{token}}",
          "type": "text"
        }
      ],
      "url": {
        "raw": "http://localhost:8080/api/blogs",
        "protocol": "http",
        "host": ["localhost"],
        "port": "8080",
        "path": ["api", "blogs"]
      }
    }
  }
 ]
}
```

Project Structure

```
src/main/java/com/blogjwt/Jwt_Blog/
│
├── Config/
│   ├── JwtAuthFilter.java      (JWT filter)
│   ├── JwtUtil.java            (Token utilities)
│   └── SecurityConfig.java     (Security configuration)
│
├── Controller/
│   └── AuthController.java     (Authentication endpoints)
│
├── DTO/
│   ├── LoginRequestDTO.java    (Login request data)
│   ├── LoginResponseDTO.java   (Login response data)
│   └── SignUpDTO.java         (Signup request data)
│
├── Entity/
│   └── User.java              (User entity)
│
├── Mapper/
│   └── AuthMapper.java        (DTO mappers)
│
├── Repository/
│   └── UserRepository.java    (JPA repository)
│
└── Service/
    └── UserService.java       (User business logic)
```

Configuration Files

application.properties

```properties
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/blog_db
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# Server
server.port=8080
```

pom.xml Dependencies

```xml
```

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

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

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

    <!-- MySQL Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>

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

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

Summary

This JWT authentication system provides:

1. **Secure User Registration**: Passwords hashed with BCrypt
2. **Token-Based Authentication**: Stateless JWT tokens
3. **Protected Endpoints**: Automatic authentication via filter
4. **RESTful API Design**: Clean separation of concerns

Key Security Features

- ✅ BCrypt password hashing
- ✅ JWT token validation
- ✅ Stateless sessions
- ✅ Filter-based authentication
- ✅ Role-based access control ready

Next Steps for Production

1. Move SECRET key to environment variables
2. Add refresh token mechanism
3. Implement proper error handling
4. Add input validation
5. Set up CORS configuration
6. Add rate limiting
7. Implement logging
8. Add unit and integration tests

Quick Reference

API Endpoints

Method	Endpoint	Auth Required	Description
POST	/auth/signup	No	Register new user
POST	/auth/login	No	Login and get token
GET	/api/*	Yes	Protected resources

Token Format

```
Authorization: Bearer <token>
```

Token Expiration

24 hours (86400000 milliseconds)

Password Requirements

No specific requirements in current implementation (should be added)

End of Documentation