# Set a **password** for your **MongoDB localhost connection**, follow these steps:

## ****🔹 1️⃣ Enable Authentication in MongoDB****

By default, MongoDB **does not require authentication** when running locally. To enforce authentication:

### **🔹 Edit MongoDB Configuration File (**mongod.conf**)**

1. Open your MongoDB configuration file:
   * **Linux/macOS:** /etc/mongod.conf
   * **Windows:** C:\Program Files\MongoDB\Server\{your\_version}\bin\mongod.cfg
2. Add or modify these lines:

security:

authorization: enabled

1. **Restart MongoDB** to apply the changes:
   * **Linux/macOS:** sudo systemctl restart mongod
   * **Windows:** Restart **MongoDB Server** from **Services**.

## ****🔹 2️⃣ Create an Admin User with a Password****

1. Open **MongoDB shell**:

mongosh

1. Switch to the admin database:

use admin

1. Create an admin user with a **username and password**:

db.createUser({

user: "admin",

pwd: "password123",

roles: [{ role: "root", db: "admin" }]

})

1. Exit the shell:

exit

## ****🔹 3️⃣ Connect to MongoDB with Authentication****

Now, you **must provide the username and password** when connecting.

### **🔹 Using MongoDB Shell**

mongosh --username admin --password password123 --authenticationDatabase admin

### **🔹 Using MongoDB Compass**

1. Open **MongoDB Compass**.
2. In the connection settings:
   * **Hostname:** localhost
   * **Port:** 27017
   * **Authentication:** Select **Username/Password**.
   * **Username:** admin
   * **Password:** password123
   * **Authentication Database:** admin
3. Click **Connect**.

### **🔹 Using Spring Boot (**application.properties**)**

Modify your **Spring Boot MongoDB configuration**:

spring.data.mongodb.uri=mongodb://admin:password123@localhost:27017/userDB?authSource=admin

## ****🚀 Final Takeaways****

✅ **Enabling authentication secures your local MongoDB.**  
✅ **You must use a username & password after enabling security.**  
✅ **Update Spring Boot, Compass, or CLI to include authentication.**

# **Environment Variables**

**Using environment variables** for sensitive credentials like **username and password** is a **best practice**. It improves **security** by keeping secrets out of your source code. 🔒✅

### **How to Use Environment Variables in Spring Boot for MongoDB?**

#### **1️⃣ Define Environment Variables in Your System**

On **Windows (Command Prompt):**

setx MONGO\_USERNAME “root”

setx MONGO\_PASSWORD “Darwin”

#### **2️⃣ Use them in** application.properties

spring.data.mongodb.username=${MONGO\_USERNAME}

spring.data.mongodb.password=${MONGO\_PASSWORD}

#### **3️⃣ (Optional) Use** application.yml **Instead**

spring:

data:

mongodb:

username: ${MONGO\_USERNAME}

password: ${MONGO\_PASSWORD}

# **How to Store Hashed Passwords in Spring Boot**

To securely store passwords in **Spring Boot**, you should **hash** them using **BCrypt**, a widely used hashing algorithm. Spring Security provides **BCryptPasswordEncoder** for this purpose.

## ****🔹 Step 1: Add Spring Security Dependency****

If you're using **Maven**, add this to your pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

For **Gradle**, add this to build.gradle:

implementation 'org.springframework.boot:spring-boot-starter-security'

## ****🔹 Step 2: Hash Passwords Before Storing****

Use BCryptPasswordEncoder to **hash** passwords before saving them.

### **Example: Hash a Password in Java**

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class PasswordHasher {

public static void main(String[] args) {

BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

String rawPassword = "mySecurePassword";

String hashedPassword = encoder.encode(rawPassword);

System.out.println("Raw Password: " + rawPassword);

System.out.println("Hashed Password: " + hashedPassword);

}

}

✅ **Output:**

Raw Password: mySecurePassword

Hashed Password: $2a$10$eP9W1kVpxeU58ZG9v6sphuYWW4a....

💡 **Every time you run it, the hash will be different** due to **salting**.

## ****🔹 Step 3: Integrate with Spring Boot User Model****

Modify your **User model** to store the hashed password:

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Document(collection = "users")

public class User {

@Id

private String id;

private String username;

private String email;

private String password; // Store the hashed password

private String role;

public User(String username, String email, String rawPassword, String role) {

this.username = username;

this.email = email;

this.password = hashPassword(rawPassword); // Hash password before storing

this.role = role;

}

private String hashPassword(String rawPassword) {

BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

return encoder.encode(rawPassword);

}

// Getters and setters

}

## ****🔹 Step 4: Hash Password Before Saving in Service Layer****

When saving a user, always hash the password.

### **Example: UserService**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.stereotype.Service;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

private BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

public User registerUser(String username, String email, String rawPassword, String role) {

String hashedPassword = passwordEncoder.encode(rawPassword); // Hash the password

User user = new User(username, email, hashedPassword, role);

return userRepository.save(user);

}

}

## ****🔹 Step 5: Verify Passwords During Login****

To compare a **raw password** with a **hashed password** (e.g., during login):

boolean isPasswordMatch = passwordEncoder.matches("enteredPassword", storedHashedPassword);

if (isPasswordMatch) {

System.out.println("Password is correct!");

} else {

System.out.println("Invalid password.");

}

## ****✅ Final Notes****

* **Never store raw passwords** in the database.
* **BCrypt is slow by design** to prevent brute-force attacks.
* **Always hash passwords before storing** in MongoDB.

# **Spring Security**

# **🔹 Step 1: How Spring Security Works by Default**

### **1️⃣ Default Behavior (Without Custom Configuration)**

* When you **add spring-boot-starter-security**, Spring Security:
  + **Secures all endpoints** (/api/\*\*, /users/\*\*, etc.).
  + Requires a **username & password** for every request.
  + Uses **Basic Authentication** (asks for username & password in Postman).
  + Generates a **default password** (Check logs when you start your app).

### **2️⃣ Where to Find the Default Password?**

When you run the application, you will see:

Using generated security password: 3b2f6b12-xxxx-xxxx-xxxx-xxxxxxxx

By default, the username is **"user"**, and the password is generated randomly.

#### **Test in Postman**

1. **Go to Postman → Authorization Tab.**
2. **Select "Basic Auth".**
3. Enter:
   * Username: **user**
   * Password: **Generated Password from Logs**
4. Send the request.

🚀 Now, your API should work because you provided authentication.

# **🔹 Step 2: How to Disable Authentication? (Only for Testing)**

If you **don’t want authentication for now**, create a **security configuration class** and allow all requests:

### **✅ Disable Authentication Completely**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth.anyRequest().permitAll()) // Allow all requests

.csrf(AbstractHttpConfigurer::disable); // Disable CSRF for testing

return http.build();

}

}

📌 **Now restart your app.** All endpoints will be open without authentication.

# **🔹 Step 3: How to Implement Authentication? (Basic Auth)**

If you want **authentication**, but with a custom username and password, follow these steps:

### **✅ 1. Define Custom Users**

Modify the SecurityConfig to define a custom username & password:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

// Define users in memory

@Bean

public InMemoryUserDetailsManager userDetailsManager() {

UserDetails admin = User.withDefaultPasswordEncoder()

.username("admin")

.password("admin123")

.roles("ADMIN")

.build();

UserDetails user = User.withDefaultPasswordEncoder()

.username("user")

.password("user123")

.roles("USER")

.build();

return new InMemoryUserDetailsManager(admin, user);

}

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN") // Only ADMIN can access /admin/\*\*

.anyRequest().authenticated() // All other endpoints require authentication

)

.httpBasic(); // Enable Basic Authentication

return http.build();

}

}

✅ **Now, restart your app.**

* If you access /admin/\*\*, it will require **"admin/admin123"**.
* Other endpoints will require **any authenticated user**.
* You **must send username/password** in **Postman → Authorization → Basic Auth**.

# **🔹 Step 4: How to Use BCrypt for Password Hashing**

You added spring-boot-starter-security to **use BCrypt** for hashing passwords. Here’s how to store and verify passwords securely.

### **✅ 1. Hash Password Before Saving to MongoDB**

Modify the Credentials class:

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class Credentials {

private String username;

private String password; // Store hashed password

public void setPassword(String rawPassword) {

BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

this.password = encoder.encode(rawPassword); // Hash password before saving

}

}

Now, when saving a user:

User user = new User();

user.getCredentials().setUsername("dk123");

user.getCredentials().setPassword("123"); // This will be hashed

### **✅ 2. Verify Password on Login**

When a user logs in, verify the password:

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public boolean verifyPassword(String rawPassword, String hashedPassword) {

BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

return encoder.matches(rawPassword, hashedPassword);

}

Example:

boolean isValid = verifyPassword("123", "$2a$10$...");

System.out.println("Password Match: " + isValid);

# **🔹 Step 5: Role-Based Authentication**

Modify SecurityConfig to **allow different roles**:

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN") // Only ADMIN can access

.requestMatchers("/user/\*\*").hasRole("USER") // Only USER can access

.anyRequest().authenticated()

)

.httpBasic();

return http.build();

}

✅ **Now, users must have the right role to access endpoints.**

* **"/admin/**" → Only ADMIN can access
* **"/user/**" → Only USER can access
* **Other endpoints** → Any authenticated user can access

# **🚀 Final Summary**

### **How Spring Security Works**

* When you add spring-boot-starter-security, it **secures all endpoints** by default.
* Without configuration, it uses **Basic Authentication** with a **generated password**.

### **How to Avoid Authentication**

* Add this to SecurityConfig:
* http.authorizeHttpRequests(auth -> auth.anyRequest().permitAll());
* This **disables authentication** for all endpoints.

### **How to Enable Authentication**

* Use SecurityConfig to:
  + Define **custom users**.
  + Use **Basic Auth**.
  + Secure specific endpoints.
  + Implement **role-based access**.

### **How to Hash Passwords**

* Use **BCrypt** to store and verify passwords securely.
* **Never store plain-text passwords** in MongoDB.

# **Spring Security Default Behavior in Your API**

Only **GET requests work, but all other requests (POST, PUT, DELETE) are blocked by default**. This happens because **Spring Security enables CSRF protection by default**.

## ****🔹 Why is Spring Blocking POST, PUT, and DELETE Requests?****

Spring Security has **Cross-Site Request Forgery (CSRF) protection enabled** by default, which **blocks state-changing requests (POST, PUT, DELETE, PATCH) unless a CSRF token is provided**.

* ✅ **GET requests are allowed** (safe operations).
* ❌ **POST, PUT, DELETE, PATCH are blocked** because they modify data and require CSRF protection.

## ****🔹 How to Fix It?****

### **1️⃣ Solution: Disable CSRF for APIs (Recommended for REST)**

Since you're building a REST API, **CSRF protection is not needed** because APIs typically use JWT tokens or API keys for security instead of browser sessions.

Modify your SecurityConfig to disable CSRF:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/users/register", "/api/users/login").permitAll() // Public endpoints

.anyRequest().authenticated() // Other endpoints require authentication

)

.csrf(AbstractHttpConfigurer::disable) // ❌ Disable CSRF for REST API

.httpBasic(); // Use Basic Authentication (Change to JWT later)

return http.build();

}

}

🔹 **Now, POST, PUT, DELETE will work!** 🚀

### **2️⃣ Alternative: Use CSRF Tokens (For Web Apps)**

If you're building a **traditional web application with forms**, you need to include a **CSRF token** in every state-changing request.

In Postman, **send a CSRF token in headers**:

1. **Make a GET request to /** (Spring Security sends a CSRF token).
2. Copy the CSRF token from the response headers (X-CSRF-TOKEN).
3. Include this token in the header for POST, PUT, DELETE requests.

## ****🚀 Final Takeaway****

* **REST APIs?** 👉 Disable CSRF (.csrf(AbstractHttpConfigurer::disable)) ✅
* **Traditional web app (Thymeleaf, JSP)?** 👉 Use CSRF tokens.

# **JWT authentication** in Spring Boot with MongoDB:

### ✅ **Step 1: Add Dependencies (**pom.xml**)**

        <dependency>

            <groupId>io.jsonwebtoken</groupId>

            <artifactId>jjwt-api</artifactId>

            <version>0.12.6</version>

        </dependency>

        <dependency>

            <groupId>io.jsonwebtoken</groupId>

            <artifactId>jjwt-impl</artifactId>

            <version>0.12.6</version>

            <scope>runtime</scope>

        </dependency>

        <dependency>

            <groupId>io.jsonwebtoken</groupId>

            <artifactId>jjwt-jackson</artifactId> <!-- Required for JSON parsing -->

            <version>0.12.6</version>

            <scope>runtime</scope>

        </dependency>

### ✅ **Step 2: Create** JwtUtil **for Token Generation & Validation**

public class JwtUtil {

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

    private static final long EXPIRATION\_TIME = 1000 \* 60 \* 60; // 1 hour

    private SecretKey key;

    // ✅ Inject secret key from application.properties

    public JwtUtil(@Value("${jwt.secret}") String secretKey) {

        this.key = Keys.hmacShaKeyFor(Decoders.BASE64.decode(secretKey));

    }

    // ✅ Validate the key after dependency injection

    @PostConstruct

    public void validateKey() {

        if (key == null) {

            throw new IllegalStateException("JWT secret key is not initialized!");

        }

        logger.info("JWT Secret Key successfully initialized.");

    }

    // ✅ Generate JWT Token

    public String generateToken(String username) {

        return Jwts.builder()

                .subject(username)

                .issuedAt(new Date())

                .expiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))

                .signWith(key)

                .compact();

    }

    // ✅ Extract Username (Optional to prevent exceptions)

    public Optional<String> extractUsername(String token) {

        try {

            return Optional.ofNullable(parseClaims(token).getSubject());

        } catch (JwtException e) {

            logger.warn("Failed to extract username: {}", e.getMessage());

            return Optional.empty();

        }

    }

    // ✅ Validate Token

    public boolean validateToken(String token, String username) {

        try {

            Claims claims = parseClaims(token);

            String extractedUsername = claims.getSubject();

            return extractedUsername.equals(username) && !isTokenExpired(claims);

        } catch (ExpiredJwtException e) {

            logger.warn("JWT Token is expired: {}", e.getMessage());

            return false;

        } catch (JwtException | IllegalArgumentException e) {

            logger.warn("Invalid JWT Token: {}", e.getMessage());

            return false;

        }

    }

    // ✅ Check Token Expiration

    private boolean isTokenExpired(Claims claims) {

        return claims.getExpiration().before(new Date());

    }

    // ✅ Common Method to Parse Claims (Avoid Code Duplication)

    private Claims parseClaims(String token) {

        return Jwts.parser()

                .verifyWith(key)

                .build()

                .parseSignedClaims(token)

                .getPayload();

    }

}

### ✅ **Step 3: Create** UserService **for User Management**

@Service

public class UserService {

    @Autowired

    private UserRepository userRepository;

    @Autowired

    private JwtUtil jwtUtil;

    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    // 1️⃣ Create User (Relies on MongoDB Unique Index for Duplicate Handling) -

    // Registration

    public User saveUser(User user) {

        // Hash the password before saving

        user.getCredentials().setPassword(passwordEncoder.encode(user.getCredentials().getPassword()));

        return userRepository.save(user); // DuplicateKeyException will be thrown if a unique constraint is violated

    }

    // 🔹 Login & Generate JWT

    public String loginUser(String username, String password) {

        User user = userRepository.findByCredentialsUsername(username)

                .orElseThrow(() -> new UserNotFoundException("Invalid credentials"));

        if (!passwordEncoder.matches(password, user.getCredentials().getPassword())) {

            throw new UserNotFoundException("Invalid credentials");

        }

        return jwtUtil.generateToken(user.getCredentials().getUsername());

        // return "jolly";

    }

    public User getUserFromToken(String token) {

        // Extract username from token

        Optional<String> extractedUsername = jwtUtil.extractUsername(token.replace("Bearer ", ""));

        if (extractedUsername.isEmpty()) {

            throw new UserNotFoundException("Invalid or expired token.");

        }

        // Fetch user from the database

        return userRepository.findByCredentialsUsername(extractedUsername.get())

                .orElseThrow(() -> new UserNotFoundException("User not found for token."));

    }

}

### ✅ **Step 4: Create** UserController **for Authentication API**

@RestController

@RequestMapping("/api/users")

public class UserController {

    @Autowired

    private UserService userService;

    // 1️⃣ Create User

    @PostMapping("/register")

    public ResponseEntity<User> createUser(@RequestBody User user) {

        User savedUser = userService.saveUser(user);

        return ResponseEntity.ok(savedUser);

    }

    // 🔹 Login & Get JWT

    @PostMapping("/login")

    public ResponseEntity<Map<String, String>> login(@RequestBody Credentials credentials) {

        String username = credentials.getUsername();

        String password = credentials.getPassword();

        String token = userService.loginUser(username, password);

        return ResponseEntity.ok(Map.of("token", token, "username", username));

    }

    @PostMapping("/dashboard")

    public ResponseEntity<Map<String, User>> dashboard(@RequestHeader("Authorization") String token) {

        User user = userService.getUserFromToken(token);

        return ResponseEntity.ok(Map.of("user", user));

    }

}

### ✅ **Step 4: Test API in Postman**

#### **1️⃣ Register User**

**Request:**  
POST http://localhost:8080/api/users/register

{

"username": "testuser",

"password": "test123"

}

**Response:**  
User registered successfully

#### **2️⃣ Login and Get Token**

**Request:**  
POST http://localhost:8080/api/users/login

{

"username": "testuser",

"password": "test123"

}

**Response:**  
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

#### **3️⃣ Access Protected Route**

**Request:**  
POST http://localhost:8080/api/users/dashboard  
**Headers:**

Authorization: Bearer <JWT\_TOKEN>

**Response:**  
403 Forbidden if not authenticated, else it will process the request.

### ✅ **Final Notes**

1. **JWT Secret Key**: Use a **strong, long, and Base64-encoded key**.
2. **Security Enhancements**: Implement a JwtFilter to validate tokens before processing requests.
3. **Password Security**: Always use BCryptPasswordEncoder for password storage.

# **JWT-based authentication for Rest-API Endpoints**

### 🔹 **Steps to Implement JWT Authentication for Endpoints**

1. **Disable Basic Auth and Default Login**
2. **Create a JwtAuthenticationFilter** to intercept requests and validate JWTs
3. **Modify SecurityConfig** to allow only JWT-authenticated users
4. **Secure endpoints with Bearer Token**

## ✅ ****1. Modify**** SecurityConfig ****to Use JWT Instead of Basic Auth****

package com.example.mongo\_integration.security;

import com.example.mongo\_integration.filter.JwtAuthenticationFilter;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.ProviderManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

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.configurers.AbstractHttpConfigurer;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration

public class SecurityConfig {

private final JwtAuthenticationFilter jwtAuthenticationFilter;

public SecurityConfig(JwtAuthenticationFilter jwtAuthenticationFilter) {

this.jwtAuthenticationFilter = jwtAuthenticationFilter;

}

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/users/register", "/api/users/login").permitAll() // Allow public access

.anyRequest().authenticated() // Secure all other endpoints

)

.csrf(AbstractHttpConfigurer::disable)

.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class); // Add JWT filter

return http.build();

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {

return authenticationConfiguration.getAuthenticationManager();

}

}

## ✅ ****2. Create**** JwtAuthenticationFilter ****to Validate JWT****

package com.example.mongo\_integration.filter;

import com.example.mongo\_integration.security.JwtUtil;

import com.example.mongo\_integration.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.security.core.userdetails.UserDetailsService;

import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component

public class JwtAuthenticationFilter extends OncePerRequestFilter {

private final JwtUtil jwtUtil;

private final UserService userService;

public JwtAuthenticationFilter(JwtUtil jwtUtil, UserService userService) {

this.jwtUtil = jwtUtil;

this.userService = userService;

}

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain) throws ServletException, IOException {

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

if (authHeader == null || !authHeader.startsWith("Bearer ")) {

filterChain.doFilter(request, response);

return;

}

String token = authHeader.substring(7);

String username = jwtUtil.extractUsername(token);

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

UserDetails userDetails = userService.loadUserByUsername(username);

if (jwtUtil.validateToken(token) && userDetails != null) {

UsernamePasswordAuthenticationToken authToken =

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

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

SecurityContextHolder.getContext().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

## ✅ ****3. Modify**** UserService ****to Load User Details****

package com.example.mongo\_integration.service;

import com.example.mongo\_integration.model.User;

import com.example.mongo\_integration.repository.UserRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.stereotype.Service;

import java.util.Optional;

@Service

public class UserService implements UserDetailsService {

@Autowired

private UserRepository userRepository;

private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

public String register(User user) {

user.setPassword(passwordEncoder.encode(user.getPassword())); // Encrypt password

userRepository.save(user);

return "User registered successfully";

}

public Optional<User> findByUsername(String username) {

return userRepository.findByUsername(username);

}

public boolean validatePassword(String rawPassword, String encodedPassword) {

return passwordEncoder.matches(rawPassword, encodedPassword);

}

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

Optional<User> user = userRepository.findByUsername(username);

return user.map(u -> org.springframework.security.core.userdetails.User

.withUsername(u.getUsername())

.password(u.getPassword())

.authorities("USER")

.build()

).orElseThrow(() -> new UsernameNotFoundException("User not found"));

}

}

## ✅ ****4. Testing JWT Authentication in Postman****

### **1️⃣ Register User**

**Request:**  
POST http://localhost:8080/api/users/register

{

"username": "testuser",

"password": "test123"

}

**Response:**  
User registered successfully

### **2️⃣ Login and Get Token**

**Request:**  
POST http://localhost:8080/api/users/login

{

"username": "testuser",

"password": "test123"

}

**Response:**

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

### **3️⃣ Access Protected Endpoint**

**Request:**  
GET http://localhost:8080/api/protected **Headers:**

Authorization: Bearer <JWT\_TOKEN>

**Response:**  
200 OK - Access Granted  
403 Forbidden - If no or invalid token is provided

## ✅ ****Which is More Secure? Spring Security Default Login vs. JWT****

|  |  |  |
| --- | --- | --- |
| Feature | Spring Security Basic Auth | JWT Authentication |
| **Security Level** | High | High (if implemented correctly) |
| **User Credentials Storage** | Stored in memory or database | Sent once during login, not stored client-side |
| **Authentication Type** | Session-based (stateful) | Token-based (stateless) |
| **Scalability** | Less scalable (server needs to maintain sessions) | Highly scalable (stateless) |
| **Use Case** | Internal APIs (Admin Panels, Enterprise Apps) | Public APIs, Microservices |

**Conclusion:**  
✅ **JWT** is more **secure for microservices** and **scalable architectures**, as it doesn't rely on server-side sessions. However, **Spring Security's default Basic Auth** is better for **small internal applications** that don’t require stateless authentication.

🚀 **Final Recommendation:** If you are developing **REST APIs or Microservices**, **JWT is the better choice**!

Let me know if you need any further modifications. 😊