# 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.