=========================== Spring Security ===========================

## ****1. Introduction to Spring Security****

### ****What is Spring Security?****

Spring Security is a framework that provides authentication, authorization, and protection against common security threats like CSRF, XSS, and session fixation. It integrates seamlessly with Spring Boot applications.

### ****Why Use Spring Security?****

* Provides built-in security mechanisms.
* Supports authentication and role-based authorization.
* Protects against common vulnerabilities like CSRF and session hijacking.
* Easy integration with OAuth2, JWT, and LDAP.

### ****If JWT is used for Stateless Authentication, then what is used for Stateful Authentication?****

For **Stateful Authentication**, the most common mechanism is **Session-Based Authentication** using **Session IDs (Cookies)**.

****Stateful Authentication and Stateless Authentication (JWT) difference and Explanation ?****

### ****Stateful Authentication (Session-Based Authentication)****

🔹 **How it Works:**

1. User logs in with credentials (username/password).
2. Server **creates a session** and stores session details in a database (or in-memory like Redis).
3. Server sends a **Session ID** (stored in a cookie) to the client.
4. On every request, the client sends the **Session ID** (cookie).
5. Server **validates the session** from its storage before processing the request.

🔹 **Technologies Used:**

* **Session IDs (JSESSIONID, PHPSESSID, etc.)**
* **HTTP Cookies**
* **In-Memory Storage (Redis, Database, etc.)**

🔹 **Example:**

* A user logs into a banking application. The session is stored on the server and linked to the user.
* When the user navigates between pages, the server validates the session using the stored data.

### ****Stateless Authentication (JWT)****

🔹 **How it Works:**

1. User logs in and gets a **JWT Token** instead of a session.
2. Token is **self-contained** (includes user info + expiry inside it).
3. No need for server-side session storage.
4. Client sends JWT in the **Authorization Header** on every request.
5. Server validates the token without checking a database.

🔹 **Technologies Used:**

* **JWT (JSON Web Token)**
* **OAuth 2.0 with JWT Tokens**
* **API Gateway Authentication**

🔹 **Example:**

* A mobile app authenticates a user using a JWT. The token is stored locally and sent with each API request.

### ****Comparison: Stateful vs. Stateless Authentication****

| **Feature** | **Stateful (Session-Based)** | **Stateless (JWT-Based)** |
| --- | --- | --- |
| Session Storage | Stored on the server (DB, Redis, etc.) | No storage on the server |
| Scalability | Less scalable (requires session sync) | Highly scalable |
| Performance | Slower (session lookup needed) | Faster (self-contained token) |
| Security | Session Hijacking risk | Token can be stolen if not secured properly |
| Use Case | Web Apps, Monoliths | APIs, Microservices, Mobile Apps |

### ****When to Use What?****

✅ **Use Stateful (Session-Based) Authentication when:**

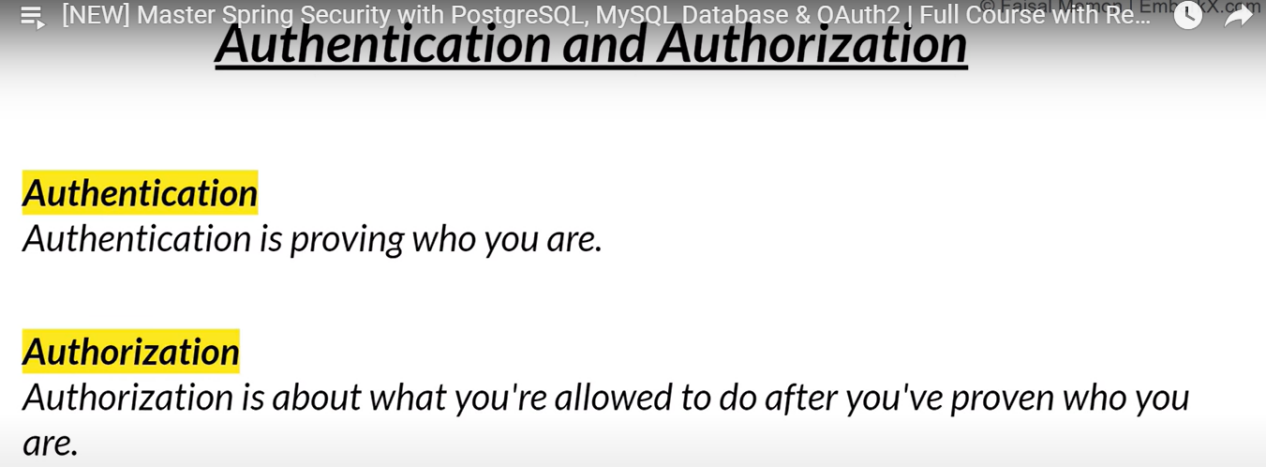
* You are developing a **traditional web application** (e.g., banking, e-commerce).
* You need **tight control over sessions** (e.g., logging out users immediately).

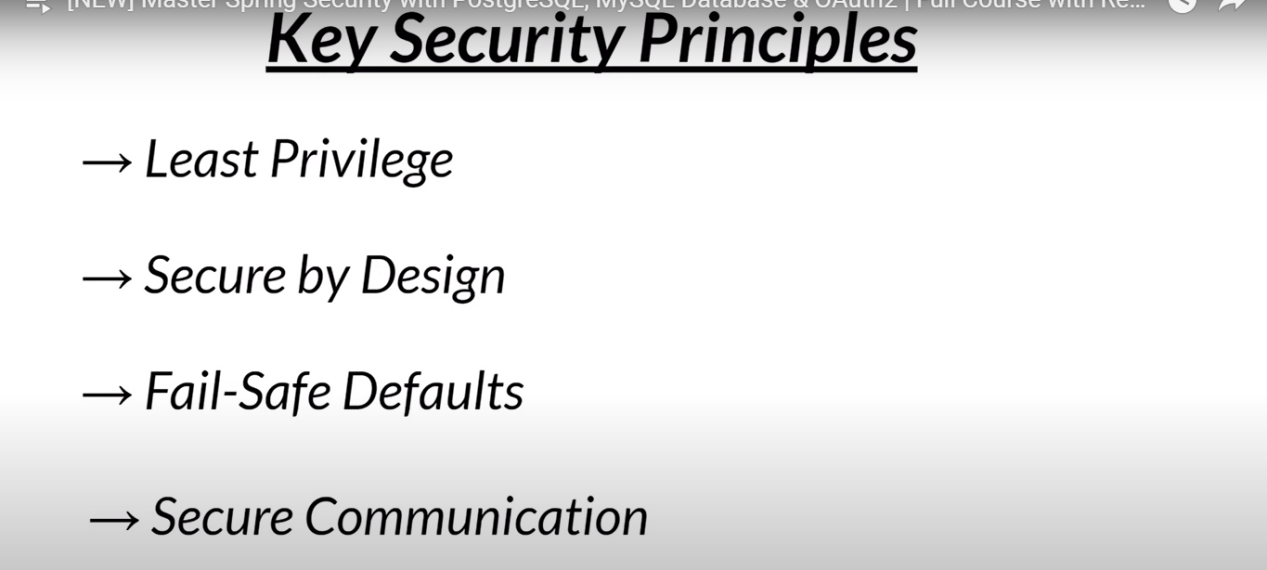
✅ **Use Stateless (JWT) Authentication when:**

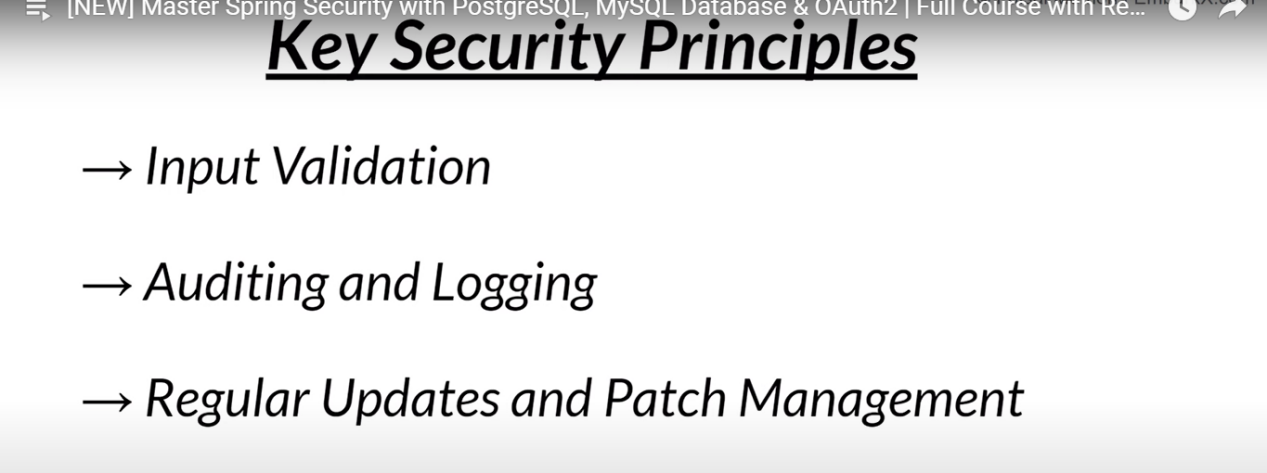
* You are building a **REST API or Microservices**.
* You need **scalability** and **performance**.
* You don't want session storage overhead.

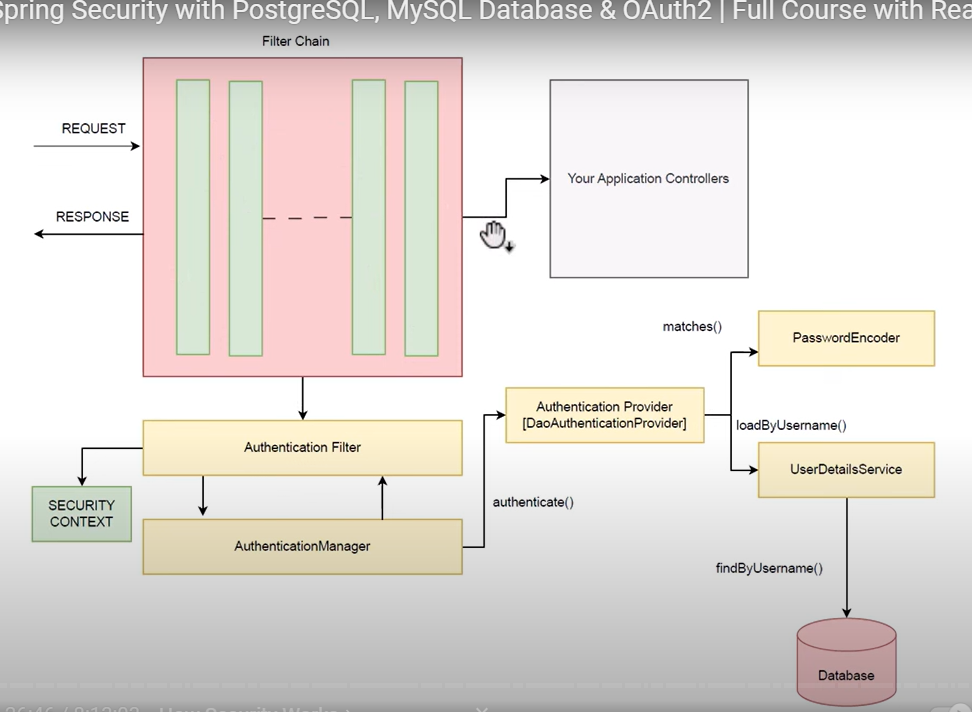
💡 **In OAuth 2.0, both approaches can be used:**

* **OAuth with Session-Based Authentication** → Stateful (e.g., login sessions for web apps).
* **OAuth with JWT Tokens** → Stateless (e.g., API authentication).









Spring Security : Mana Project lo Spring security **Dependencie** Add chesthe by default ga authentication page anedi vasthundi -- dinni manam coustmize chesukoni use chesukovali and also spring sequriry anedi by default ga session management anedi kuda chesthundi

<dependency>

<groupId>org.springframework.security</groupId>

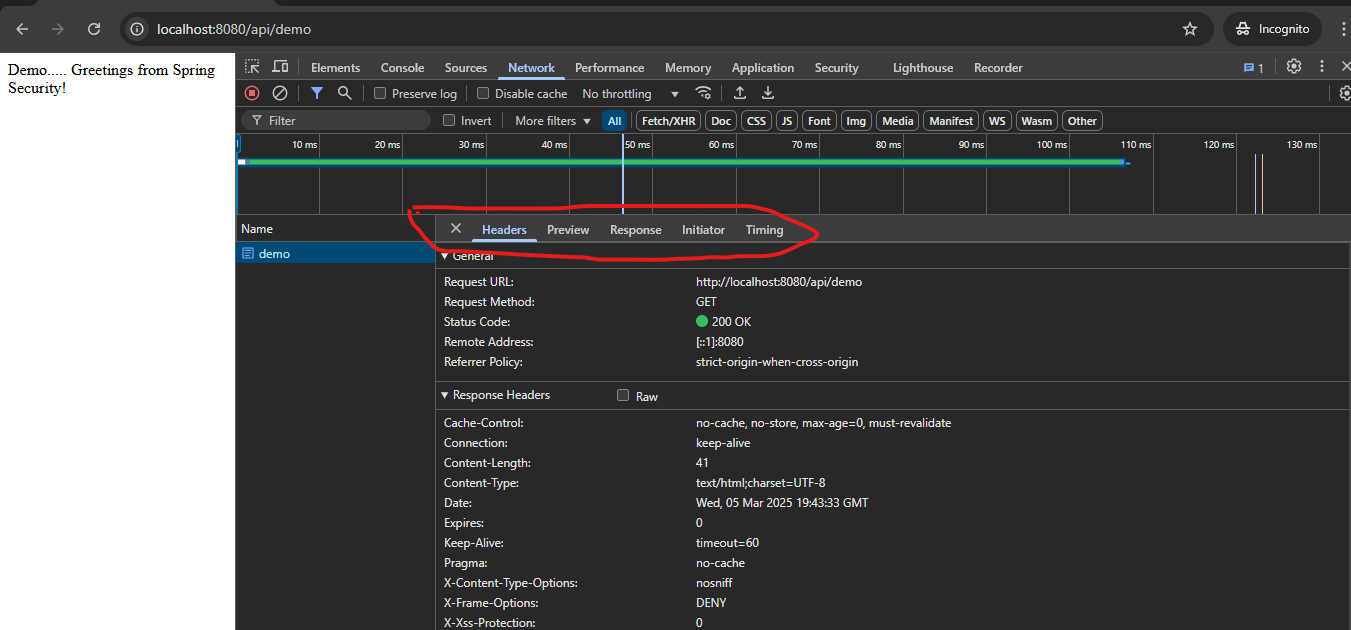
<artifactId>spring-security-test</artifactId>

<scope>test</scope>

</dependency>

What will happened if use STATELESS session in our project?

STATELESS vadithe session id anedi server lo store avvadu and every rest-end point request anedi every time authentication avuthundi



Normal ga sesstion(Cookies) anedi oka ID genarate avvali but avvaledu

http.sessionManagement(session ->

session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

: below things will happened when we use STATELESS

1. **User Can Log In** ✅
   1. The user can still authenticate (e.g., using a username/password with JWT/OAuth).
2. **No Session Storage** ❌
   1. Spring Security **will not create or use HTTP sessions** (JSESSIONID will not be stored).
3. **Every Request Must Be Authenticated** 🔁
   1. Since there is no session, the client must send credentials (e.g., JWT token) **in every request**.
   2. The server will validate the token on each request before granting access.

### ****Example Flow with**** STATELESS ****(Using JWT)****

1. **User Logs In** → Sends credentials to /login.
2. **Server Generates a JWT** → Returns it in the response.
3. **Client Stores JWT** → Typically in localStorage or sessionStorage (not cookies).
4. **Client Sends JWT with Every Request** →

GET /user/profile

Authorization: Bearer <JWT-TOKEN>

1. **Server Validates JWT** → Grants or denies access.

### ****Key Differences Compared to Stateful Sessions****

| **Feature** | **Stateful (IF\_REQUIRED)** | **Stateless (STATELESS)** |
| --- | --- | --- |
| Session Created? | ✅ Yes (JSESSIONID cookie) | ❌ No |
| Authenticated Once? | ✅ Yes (until logout) | ❌ No (every request must authenticate) |
| Scalable for APIs? | ❌ Not ideal | ✅ Yes, fully scalable |
| Token Storage | Session on server | JWT/OAuth token in client |

### ****When Should You Use**** STATELESS****?****

✅ REST APIs, Microservices, Mobile Apps, JWT Authentication, Scalable Systems.

❌ If you need **session tracking** (e.g., traditional login/logout in web apps), then **do not use** STATELESS.

Spring Sequrity some IMP default class and Interfaces usage of those class Example : User,Userdetails,userdetailsservices,userdetailsManger,InMemoryUserDetailsManager ?

### ****Spring Security Core Authentication Components****

These classes and interfaces are fundamental in **Spring Security’s authentication system**. They help manage user details, authentication, and storage mechanisms.

## ****1️⃣**** User ****(Concrete Class)****

### ****Role:**** Represents a user in Spring Security.

* It is a **default implementation of** UserDetails.
* Used to create users with **username, password, and roles**.

### ****✅ Example Usage****

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

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

UserDetails user = User.withUsername("john")

.password("{noop}password") // `{noop}` means no password encoding

.roles("USER") // Adds ROLE\_USER authority

.build();

### ****🔹 What it does?****

* Defines user credentials (username, password, roles).
* Typically used when manually creating user details.
* Stores authorities (ROLE\_USER).

## ****2️⃣**** UserDetails ****(Interface)****

### ****Role:**** Represents the structure of a user in Spring Security.

* Defines methods like:
  + getUsername()
  + getPassword()
  + getAuthorities()
  + isAccountNonExpired()
  + isAccountNonLocked()
  + isCredentialsNonExpired()
  + isEnabled()

### ****✅ Example (Custom Implementation)****

import org.springframework.security.core.GrantedAuthority;

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

import java.util.Collection;

import java.util.List;

public class CustomUserDetails implements UserDetails {

private String username;

private String password;

private List<GrantedAuthority> authorities;

public CustomUserDetails(String username, String password, List<GrantedAuthority> authorities) {

this.username = username;

this.password = password;

this.authorities = authorities;

}

@Override

public String getUsername() { return username; }

@Override

public String getPassword() { return password; }

@Override

public Collection<? extends GrantedAuthority> getAuthorities() { return authorities; }

@Override

public boolean isAccountNonExpired() { return true; }

@Override

public boolean isAccountNonLocked() { return true; }

@Override

public boolean isCredentialsNonExpired() { return true; }

@Override

public boolean isEnabled() { return true; }

}

### ****🔹 What it does?****

* Defines how user details are structured.
* Custom implementations are required when fetching users from **database (JPA, JDBC, etc.).**

## ****3️⃣**** UserDetailsService ****(Interface)****

### ****Role:**** Loads user details from ****any source (DB, in-memory, LDAP, etc.).****

* Used by Spring Security for authentication.

### ****✅ Example Usage****

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

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

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

import java.util.Optional;

public class CustomUserDetailsService implements UserDetailsService {

private final UserRepository userRepository; // Assume a JPA Repository

public CustomUserDetailsService(UserRepository userRepository) {

this.userRepository = userRepository;

}

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

return userRepository.findByUsername(username)

.map(user -> new CustomUserDetails(user.getUsername(), user.getPassword(), user.getAuthorities()))

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

}

}

### ****🔹 What it does?****

* Loads user details from **database or another source**.
* Used internally by Spring Security to **validate user authentication**.

## ****4️⃣**** UserDetailsManager ****(Interface)****

### ****Role:**** Extension of UserDetailsService, allows user ****creation, updating, deletion****.

* It adds methods like:
  + createUser(UserDetails user)
  + updateUser(UserDetails user)
  + deleteUser(String username)
  + changePassword(String oldPassword, String newPassword)

### ****✅ Example Usage****

import org.springframework.security.provisioning.UserDetailsManager;

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

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

public class CustomUserDetailsManager implements UserDetailsManager {

@Override

public void createUser(UserDetails user) {

// Logic to save user to DB

}

@Override

public void updateUser(UserDetails user) {

// Logic to update user in DB

}

@Override

public void deleteUser(String username) {

// Logic to delete user from DB

}

@Override

public void changePassword(String oldPassword, String newPassword) {

// Logic to change user password

}

@Override

public boolean userExists(String username) {

// Logic to check if user exists in DB

return false;

}

@Override

public UserDetails loadUserByUsername(String username) {

return null; // Logic to load user from DB

}

}

### ****🔹 What it does?****

* Handles **CRUD operations** for users.
* Used when an application needs **user management functionality**.

## ****5️⃣**** InMemoryUserDetailsManager ****(Concrete Class)****

### ****Role:**** Stores user details ****in-memory**** (not DB).

* Useful for testing or simple applications.

### ****✅ Example Usage****

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

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

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

UserDetails user = User.withUsername("admin")

.password("{noop}password") // No password encoding

.roles("ADMIN")

.build();

UserDetailsManager userDetailsManager = new InMemoryUserDetailsManager(user);

### ****🔹 What it does?****

* Stores users **in-memory (RAM)**.
* **Not recommended for production** (use JdbcUserDetailsManager or custom UserDetailsService).

# ****Other Important Classes & Interfaces in Spring Security****

Besides the above, you should also know:

### ****6️⃣**** GrantedAuthority ****(Interface)****

### ****Role:**** Represents user ****permissions/roles****.

* Used inside UserDetails.getAuthorities().

**Example:**

import org.springframework.security.core.GrantedAuthority;

public class CustomAuthority implements GrantedAuthority {

private String authority;

public CustomAuthority(String authority) { this.authority = authority; }

@Override

public String getAuthority() { return authority; }

}

### ****7️⃣**** Authentication ****(Interface)****

### ****Role:**** Represents ****current logged-in user’s authentication details****.

**Example (Get Current User Details in Controller):**

import org.springframework.security.core.Authentication;

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

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

@RestController

public class UserController {

@GetMapping("/user")

public String getUser(Authentication authentication) {

return "Logged in user: " + authentication.getName();

}

}

### ****8️⃣**** AuthenticationManager ****(Interface)****

### ****Role:**** ****Handles authentication logic.****

* Takes Authentication as input and validates credentials.

**Example:**

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.AuthenticationException;

public class CustomAuthManager implements AuthenticationManager {

@Override

public Authentication authenticate(Authentication authentication) throws AuthenticationException {

String username = authentication.getName();

String password = authentication.getCredentials().toString();

if ("admin".equals(username) && "password".equals(password)) {

return new UsernamePasswordAuthenticationToken(username, password);

}

throw new AuthenticationException("Invalid credentials") {};

}

}

### ****Summary Table****

| **Component** | **Role** |
| --- | --- |
| User | Prebuilt user implementation (UserDetails) |
| UserDetails | Defines user structure (username, password, roles) |
| UserDetailsService | Loads users from DB or other sources |
| UserDetailsManager | Manages users (CRUD operations) |
| InMemoryUserDetailsManager | Stores users in memory (not recommended for production) |
| GrantedAuthority | Represents user permissions/roles |
| Authentication | Represents current user session |
| AuthenticationManager | Handles authentication |

### ****📝 Conclusion****

* If **storing users in memory**, use InMemoryUserDetailsManager.
* If **storing users in DB**, implement UserDetailsService.
* If **you need user management (create/update/delete)**, use UserDetailsManager.

**if we have our own user details table means then what is the use of UserDetails, in my custom table name "LoginUsers" and custom columns i have then what is the use of this class UserDetails ?**

Great question! If you have a **custom user table (**LoginUsers**)** with custom columns, you might wonder:

**"Why should I implement** UserDetails**?"**

### ****Why**** UserDetails ****is Needed?****

* Spring Security expects **authentication logic** to follow a standard structure.
* UserDetails provides **a contract (interface)** for defining user attributes **in a format Spring Security understands**.
* Even if your table has different column names, UserDetails **adapts** your custom user structure to Spring Security.

## ****✅ Example: Custom**** LoginUsers ****Table****

Let's say your table has **different column names** than what Spring Security expects:

| **Column Name** | **Description** |
| --- | --- |
| id | Primary key |
| user\_email | User's email (used as username) |
| user\_password | Encrypted password |
| is\_active | Whether the user is active |
| role | User role (e.g., ADMIN, USER) |

## ****Step 1️⃣: Create**** LoginUsers ****Entity (JPA Example)****

If you're using **Spring Data JPA**, your entity might look like this:

import jakarta.persistence.\*;

import lombok.Getter;

import lombok.Setter;

@Entity

@Getter

@Setter

@Table(name = "LoginUsers")

public class LoginUsers {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(unique = true, nullable = false)

private String userEmail;

@Column(nullable = false)

private String userPassword;

private boolean isActive;

private String role; // Can store values like "ADMIN" or "USER"

}

## ****Step 2️⃣: Implement**** UserDetails

* Since LoginUsers **does not match** Spring Security's UserDetails format,  
  we create a CustomUserDetails class **to bridge the gap**.

import org.springframework.security.core.GrantedAuthority;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

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

import java.util.Collection;

import java.util.List;

public class CustomUserDetails implements UserDetails {

private LoginUsers user; // Store our custom user object

public CustomUserDetails(LoginUsers user) {

this.user = user;

}

@Override

public String getUsername() {

return user.getUserEmail(); // Map custom column to Spring Security format

}

@Override

public String getPassword() {

return user.getUserPassword(); // Map custom password column

}

@Override

public Collection<? extends GrantedAuthority> getAuthorities() {

return List.of(new SimpleGrantedAuthority("ROLE\_" + user.getRole())); // Convert role to Spring format

}

@Override

public boolean isAccountNonExpired() {

return true; // Modify if you track expiration

}

@Override

public boolean isAccountNonLocked() {

return true; // Modify if you track account locks

}

@Override

public boolean isCredentialsNonExpired() {

return true; // Modify if you track credential expiration

}

@Override

public boolean isEnabled() {

return user.isActive(); // Use your custom column

}

}

🔹 **What this does:**  
✅ **Maps** LoginUsers **fields to Spring Security format.**  
✅ **Tells Spring Security how to retrieve user details.**

## ****Step 3️⃣: Implement**** UserDetailsService ****to Load Users from Database****

Now, Spring Security needs a way to **fetch users from the database**.  
We use UserDetailsService to implement this logic.

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;

@Service

public class CustomUserDetailsService implements UserDetailsService {

private final LoginUsersRepository userRepository; // Assume JPA Repository

public CustomUserDetailsService(LoginUsersRepository userRepository) {

this.userRepository = userRepository;

}

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

LoginUsers user = userRepository.findByUserEmail(username)

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

return new CustomUserDetails(user);

}

}

🔹 **What this does:**  
✅ Fetches user from LoginUsers table.  
✅ Converts it into UserDetails using CustomUserDetails.  
✅ Returns the user details to Spring Security for authentication.

## ****Step 4️⃣: Configure Spring Security****

Now, tell Spring Security to use CustomUserDetailsService for authentication.

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.dao.DaoAuthenticationProvider;

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

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

import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration

public class SecurityConfig {

@Bean

public UserDetailsService userDetailsService(LoginUsersRepository repository) {

return new CustomUserDetailsService(repository);

}

@Bean

public AuthenticationManager authenticationManager(UserDetailsService userDetailsService) {

DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService);

authProvider.setPasswordEncoder(passwordEncoder());

return new ProviderManager(authProvider);

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder(); // Use bcrypt for secure passwords

}

}

🔹 **What this does:**  
✅ Registers CustomUserDetailsService in Spring Security.  
✅ Uses BCryptPasswordEncoder to hash passwords securely.  
✅ Defines authentication logic.

## ****Summary****

| **Component** | **Why It’s Needed** |
| --- | --- |
| UserDetails | Converts custom LoginUsers table into Spring Security format |
| UserDetailsService | Fetches user details from DB and returns UserDetails |
| AuthenticationManager | Manages authentication logic |
| PasswordEncoder | Secures passwords with hashing (e.g., bcrypt) |

## ****Why Not Just Use**** LoginUsers ****Directly?****

**Problem:** Spring Security expects UserDetails, but LoginUsers has different column names and no getAuthorities().  
**Solution:** Implement UserDetails to make LoginUsers **compatible with Spring Security**.

## ****Final Takeaway****

If you have a **custom user table** (LoginUsers), you **must**: 1️⃣ Implement UserDetails to map your table fields.  
2️⃣ Implement UserDetailsService to load users from DB.  
3️⃣ Configure authentication with AuthenticationManager.

Creating users by using H2-file based data base but while creating I am facing errorr   
Caused by: org.h2.jdbc.JdbcSQLSyntaxErrorException: Table "USERS" not found (this database is empty); SQL statement:

Table anedi not found ani vasthundi but nenu Entity class kakunda table Script ni Project lo E loaction lo Create chesanu src/main/resources/schema.sql

But a table Script anedi Springboot ki theliyatam ledu?

Sloustion :

Yes, I fully understand your requirement now — you want to **insert user details dynamically via code** (not data.sql) into a **file-based H2 database**.

✅ **Key Requirement:** Use JdbcUserDetailsManager to insert user details dynamically from **Java code** (not via data.sql).  
✅ **Database Type:** H2 (File-Based).

### ****Why Did I Mention**** data.sql ****Earlier?****

I initially suggested data.sql as an option because:

* Your code inserts users like "user" and "admin" — which is often static data, typically handled by data.sql.
* Using data.sql helps reduce redundant logic in your code for predefined users.

However, since your requirement is to achieve this **via code only**, let's proceed with that approach.

### ****Correct Approach for Dynamic Insertion Using**** JdbcUserDetailsManager

The main issue in your code is:

❌ **User Insertion Logic Executes Before H2 Table Creation.**  
✅ We need to ensure that the table is created **before** inserting users.

### ****Step 1: Correct**** application.properties ****Configuration****

spring.application.name=Spring\_SecurityWith\_H2

# H2 Database (File-Based)

spring.datasource.url=jdbc:h2:file:./data/security\_db

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# Ensure Schema Creation

spring.sql.init.mode=always

spring.sql.init.schema-locations=classpath:schema.sql

### ****Step 2: Create**** schema.sql ****for Table Creation****

Since JdbcUserDetailsManager expects a specific table structure, you must create it.

src/main/resources/schema.sql

CREATE TABLE IF NOT EXISTS USERS (

USERNAME VARCHAR(50) NOT NULL PRIMARY KEY,

PASSWORD VARCHAR(500) NOT NULL,

ENABLED BOOLEAN NOT NULL

);

CREATE TABLE IF NOT EXISTS AUTHORITIES (

USERNAME VARCHAR(50) NOT NULL,

AUTHORITY VARCHAR(50) NOT NULL,

FOREIGN KEY (USERNAME) REFERENCES USERS(USERNAME)

);

CREATE UNIQUE INDEX IF NOT EXISTS IX\_AUTH\_USERNAME ON AUTHORITIES (USERNAME, AUTHORITY);

### ****Step 3: Modify**** SecurityConfig ****to Insert Users via Code****

Now modify your userDetailsService() method to insert users dynamically after table creation.

✅ Add a CommandLineRunner to delay user insertion until the application context is ready.

**Updated** SecurityConfig

@Configuration

@EnableWebSecurity

@EnableMethodSecurity

public class SecurityConfig {

@Autowired

private DataSource dataSource;

@Bean

public UserDetailsService userDetailsService() {

return new JdbcUserDetailsManager(dataSource);

}

@Bean

public CommandLineRunner insertDefaultUsers(UserDetailsService userDetailsService) {

return args -> {

JdbcUserDetailsManager jdbcUserDetailsManager = (JdbcUserDetailsManager) userDetailsService;

if (!jdbcUserDetailsManager.userExists("user")) {

UserDetails user1 = User.withUsername("user")

.password("{noop}password")

.roles("USER")

.build();

UserDetails user2 = User.withUsername("admin")

.password("{noop}password")

.roles("ADMIN")

.build();

jdbcUserDetailsManager.createUser(user1);

jdbcUserDetailsManager.createUser(user2);

System.out.println("Users inserted successfully!");

}

};

}

@Bean

@Order(SecurityProperties.BASIC\_AUTH\_ORDER)

SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

http.authorizeHttpRequests((requests) ->

requests.requestMatchers("/h2-console/\*\*").permitAll()

.anyRequest().authenticated());

http.sessionManagement(session ->

session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

http.httpBasic(withDefaults());

http.headers(headers ->

headers.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin));

http.csrf(AbstractHttpConfigurer::disable);

return http.build();

}

}

### ****Step 4: Key Explanation****

✅ The CommandLineRunner ensures user creation happens **after the database schema is ready**.  
✅ The .userExists() check prevents duplicate entries.  
✅ No need for data.sql — everything is dynamic.

### ****Step 5: Testing****

1. Run the application.
2. Access <http://localhost:8080/h2-console>.
   * **JDBC URL:** jdbc:h2:file:./data/security\_db
   * **Username:** sa
   * **Password:** (leave empty)
3. Verify the USERS and AUTHORITIES tables are populated.

### ****Expected Outcome****

✅ Users "user" and "admin" will be dynamically inserted via code.  
✅ Data will persist in your **file-based H2 database** even after restarting the application.

### ****Final Notes****

* This method ensures that your requirements are fully met — **file-based H2** with **dynamic user insertion via code**.
* If any issues persist, enabling debug mode can provide additional insights:

logging.level.org.springframework=DEBUG

but if i use in-memory means not need CommandLineRunner right ?

Exactly! If you use an **in-memory H2 database**, you **don't need** the CommandLineRunner because:

✅ **In-memory H2 database** is created instantly when the application starts.  
✅ The JdbcUserDetailsManager can directly insert data during bean creation since the schema is created immediately.

### ****Key Difference: Why**** CommandLineRunner ****for File-Based H2 Only?****

🔹 **In-Memory H2:**

* Schema creation happens automatically during startup.
* The userDetailsService() method can safely insert users without timing issues.
* No CommandLineRunner required.

🔹 **File-Based H2:**

* The database file is created during startup, but tables may not exist when userDetailsService() tries to insert users.
* To ensure the table is ready before inserting users, we use CommandLineRunner.

### ****Example for In-Memory H2 Configuration****

application.properties

spring.application.name=Spring\_SecurityWith\_H2

# In-Memory H2 Configuration

spring.datasource.url=jdbc:h2:mem:security\_db

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# Ensure Schema Creation (optional but useful)

spring.sql.init.mode=always

spring.sql.init.schema-locations=classpath:schema.sql

### ****In-Memory Code Without**** CommandLineRunner

In this case, the userDetailsService() itself can handle user creation directly:

SecurityConfig **(In-Memory Version)**

@Configuration

@EnableWebSecurity

@EnableMethodSecurity

public class SecurityConfig {

@Autowired

private DataSource dataSource;

@Bean

public UserDetailsService userDetailsService() {

JdbcUserDetailsManager savinguser = new JdbcUserDetailsManager(dataSource);

if (!savinguser.userExists("user")) {

UserDetails user1 = User.withUsername("user")

.password("{noop}password")

.roles("USER")

.build();

UserDetails user2 = User.withUsername("admin")

.password("{noop}password")

.roles("ADMIN")

.build();

savinguser.createUser(user1);

savinguser.createUser(user2);

}

return savinguser;

}

@Bean

@Order(SecurityProperties.BASIC\_AUTH\_ORDER)

SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

http.authorizeHttpRequests((requests) ->

requests.requestMatchers("/h2-console/\*\*").permitAll()

.anyRequest().authenticated());

http.sessionManagement(session ->

session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

http.httpBasic(withDefaults());

http.headers(headers ->

headers.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin));

http.csrf(AbstractHttpConfigurer::disable);

return http.build();

}

}

### ****Summary — When to Use What?****

✅ **In-Memory H2:** → No CommandLineRunner needed.  
✅ **File-Based H2:** → Use CommandLineRunner to avoid timing issues during startup.

previously u mentioned file based h2 db means we need to create locally one folder for h2 file based storage right ?C:/h2-database/security\_h2 like this whether is correct or not?

because now u provided with out local folder spring.datasource.url=jdbc:h2:file:./data/security\_db

Yes, you're absolutely correct! For **file-based H2 database**, you must specify a **file path** to ensure the data is stored persistently on your local system.

### ****File-Based H2 Database Configuration — Correct Approach****

If you want to use **file-based H2**, you need to specify a **local folder path** like this:

application.properties

spring.application.name=Spring\_SecurityWith\_H2

# File-Based H2 Database Configuration

spring.datasource.url=jdbc:h2:file:C:/h2-database/security\_h2

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

# H2 Console

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# SQL Initialization

spring.sql.init.mode=always

spring.sql.init.schema-locations=classpath:schema.sql

spring.sql.init.data-locations=classpath:data.sql

✅ jdbc:h2:file:C:/h2-database/security\_h2 → Saves your H2 database in the C:/h2-database folder.  
✅ Make sure the folder C:/h2-database exists, or H2 will attempt to create it automatically.

### ****Alternative Path (Relative to Project)****

If you prefer to store the H2 file inside your project folder, you can use:

spring.datasource.url=jdbc:h2:file:./data/security\_db

✅ ./data/security\_db → Saves the H2 database in the /data folder inside your project directory.  
✅ The folder will be created automatically if it doesn't exist.

### ****Key Differences****

| **Aspect** | **File-Based H2** | **In-Memory H2** |
| --- | --- | --- |
| **Data Persistence** | Data persists after app shutdown. | Data is lost after app shutdown. |
| **Requires Folder Creation?** | Yes, manually create the folder. | No, managed entirely in memory. |
| **URL Format** | jdbc:h2:file:C:/path/to/db | jdbc:h2:mem:db\_name |
| **Performance** | Slightly slower (disk I/O) | Faster (entirely in memory). |

### ****Recommendation for Your Case****

Since you need **persistent** data with H2 (file-based), use this approach:

1. Create the folder: C:/h2-database
2. Use this property:

spring.datasource.url=jdbc:h2:file:C:/h2-database/security\_h2

This ensures your data will remain even after restarting the application.

Eppati daka manaki thelisina topic:

* H2 file based vadali ante local ga folder create cheyali anedi
* A next with out folder kuda spring boot project lone file ni create chesi use cheyachhu anedi thesuluskunnam
* Spring security lo H2 file based vadali ante CommandLineRunner Manditory anukunnam
* But CommandLineRunner lekunda kuda user ni H2 file based db lo rayachhu   
    
  Nenu CommandLineRunner ni use chesi code rase tappudu same users table not fond error ne vacchindi, but with out CommandLineRunner use chesinappudu errro raledu?

Let's clear the confusion step by step to ensure you have complete clarity on when and why CommandLineRunner is required — and when it’s **not**.

## ****Key Differences Between In-Memory H2 and File-Based H2****

### 1️⃣ ****In-Memory H2 Database****

✅ The database is created in **RAM** and initialized immediately during startup.  
✅ **Tables are ready instantly** once the application starts.  
✅ JdbcUserDetailsManager can insert data directly in the userDetailsService() method itself.  
✅ CommandLineRunner **is NOT required.**

### 2️⃣ ****File-Based H2 Database****

✅ The database is created as a **physical file** (e.g., C:/h2-database/security\_h2).  
✅ While the database file is created immediately, the **tables may not exist yet** when the userDetailsService() method runs.  
✅ Since userDetailsService() might execute **before the schema is created**, any user insertion logic might fail with **"Table Not Found"** errors.  
✅ CommandLineRunner **is recommended** in this case to ensure data is inserted **after** the schema is initialized.

## 🚨 ****Why Was**** CommandLineRunner ****Suggested Earlier?****

In the earlier conversation, your configuration was:

spring.datasource.url=jdbc:h2:file:C:/h2-database/security\_h2

This is a **file-based H2 configuration**. Since tables in file-based H2 might take a moment to initialize, there’s a risk that the userDetailsService() method runs **before** tables are created.

To prevent this race condition, I suggested CommandLineRunner because it runs **after the application context is fully initialized** — ensuring that the tables exist before inserting data.

## ✅ ****Why Does Your Latest Code Work Without**** CommandLineRunner****?****

In your updated code, you are using:

spring.datasource.url=jdbc:h2:file:./data/security\_db

Even though this is still **file-based**, two key factors might explain why it's working without CommandLineRunner:

**Order of Execution**:

* 1. The JdbcUserDetailsManager logic in your userDetailsService() method is getting executed **after** the schema is created.
  2. This may happen because Spring Boot is handling the schema initialization slightly earlier in the lifecycle than before.

**Database Behavior in Latest H2 Versions**:

* 1. Recent versions of H2 (like 2.3.232) have improved internal mechanics where table creation and file-based H2 initialization happen faster, reducing the risk of timing issues.

## ****Best Practice for Production Code****

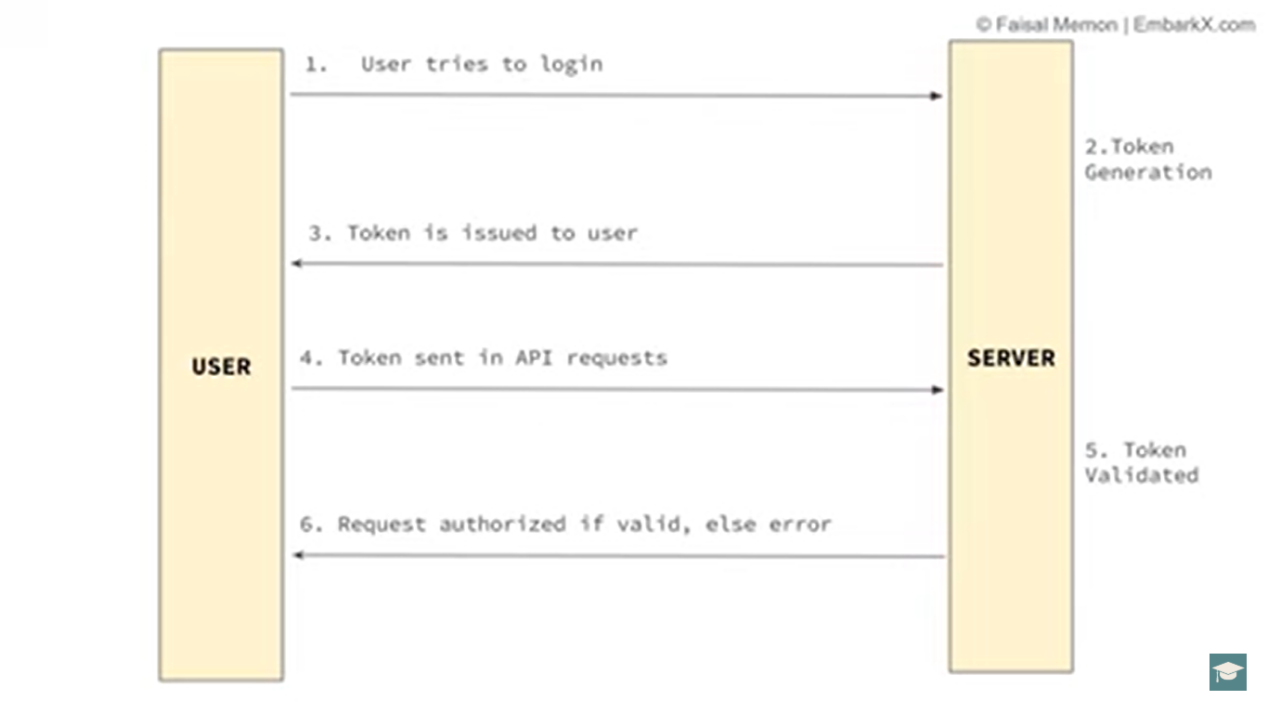
Even if your latest code works without CommandLineRunner, using it in **file-based H2** is still considered a **best practice** to ensure predictable behavior across different environments or versions.

If you're confident your current setup is stable without CommandLineRunner, it’s perfectly fine — no need to add extra logic unless you face timing issues.

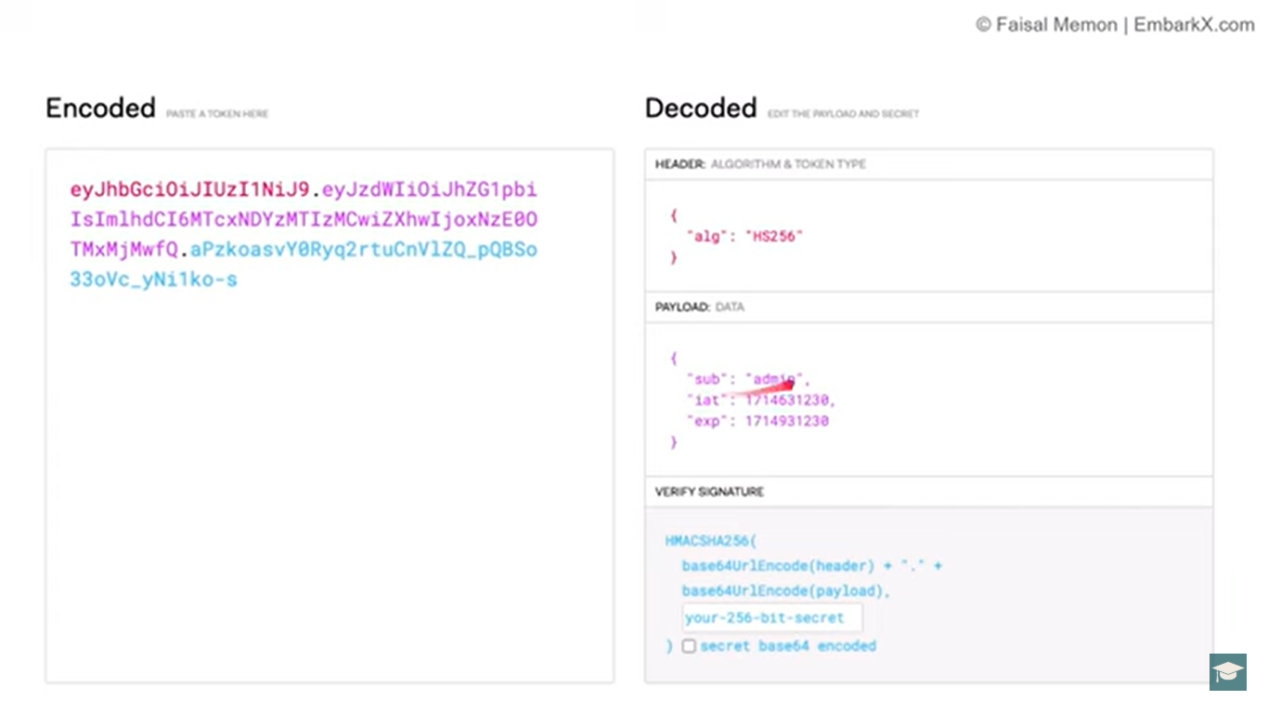
## 💡 ****Final Recommendation****

* **In-Memory H2** ➔ ✅ No CommandLineRunner required
* **File-Based H2** ➔ 🔎 **Recommended** to use CommandLineRunner for guaranteed stability
* If your current code is working fine without CommandLineRunner, you can **proceed as is**, but be aware that timing issues may arise in some environments.

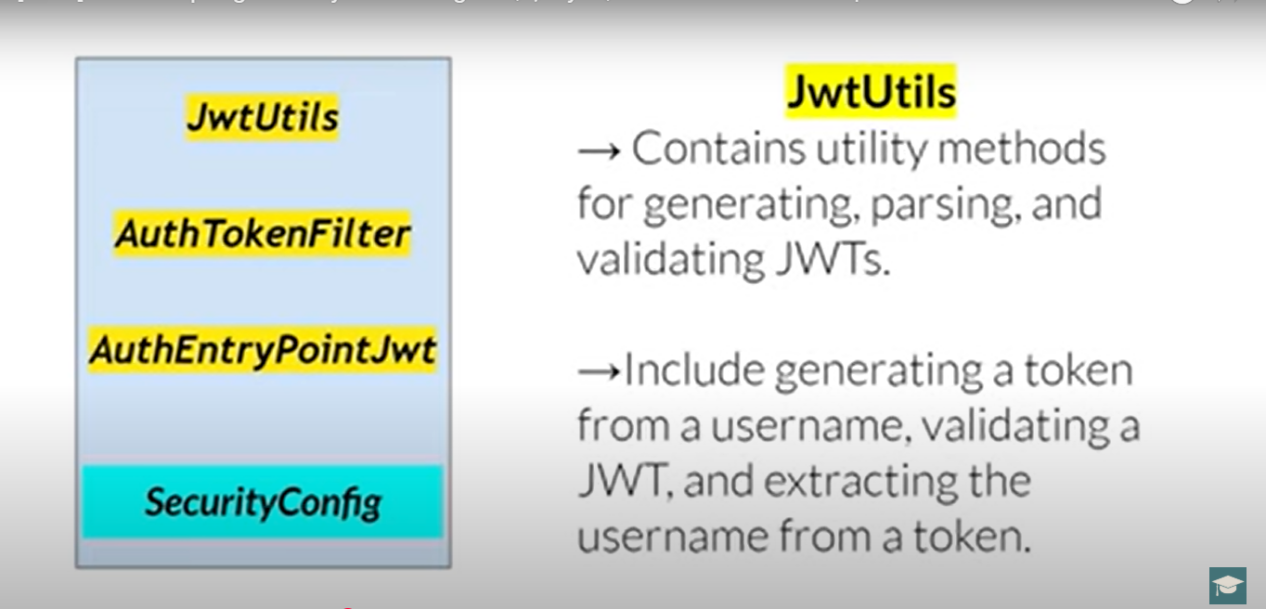
**========================== JWT Token ============================**

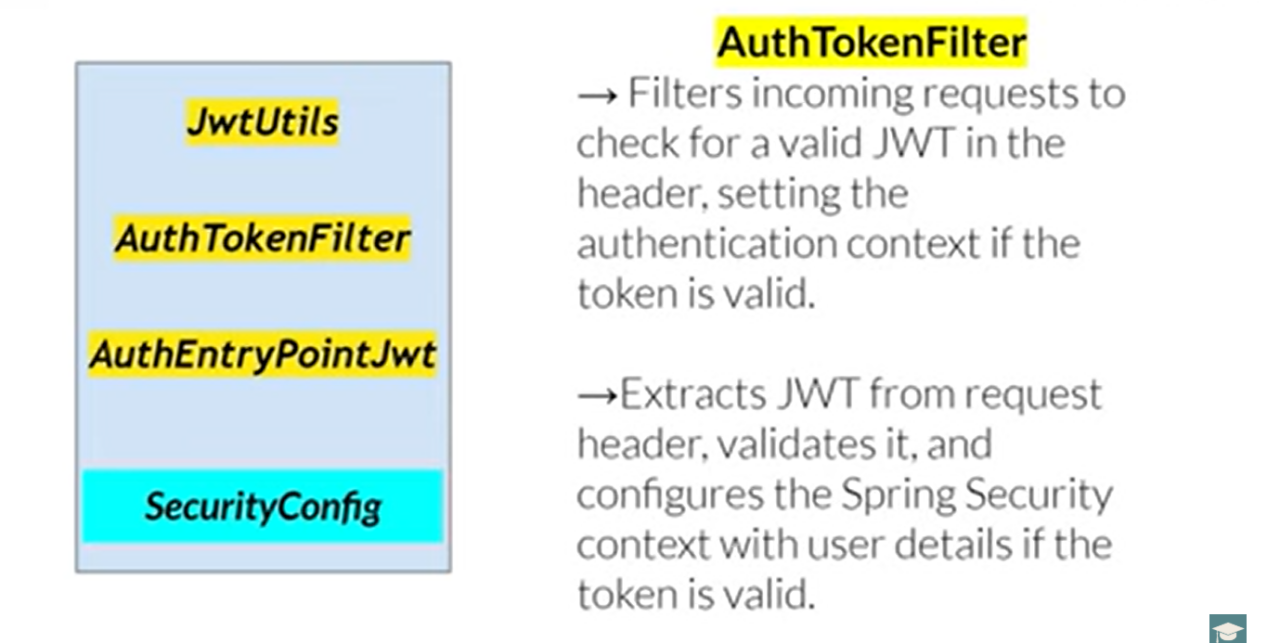


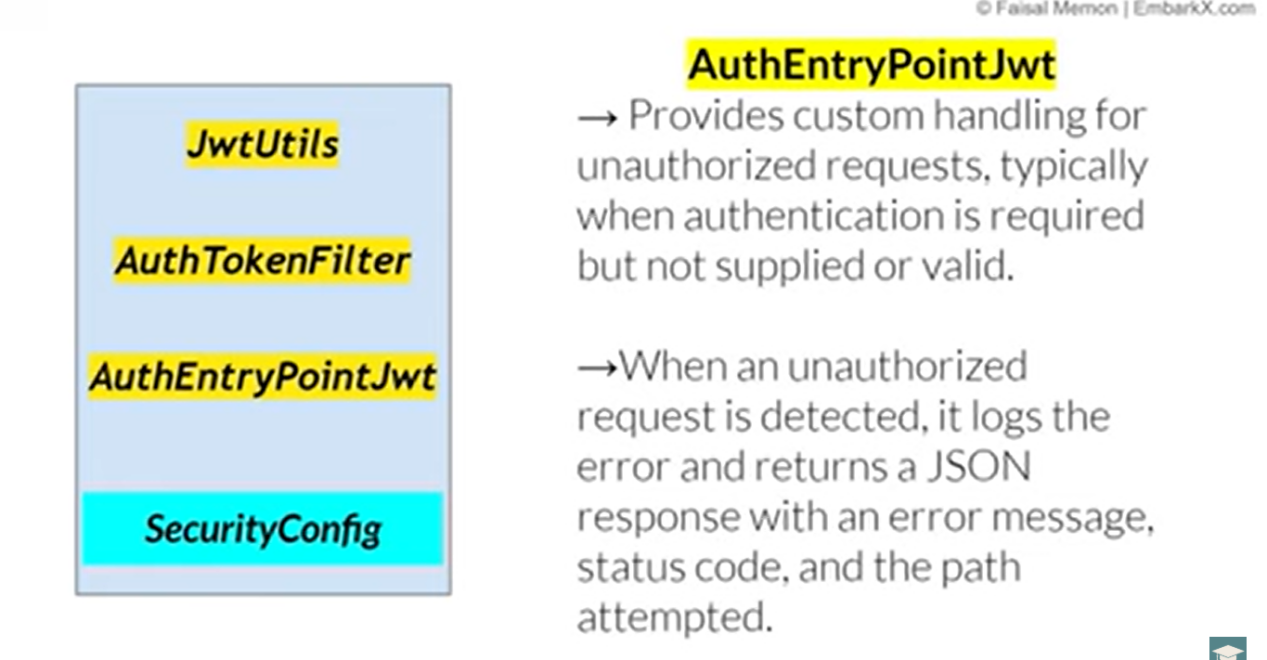


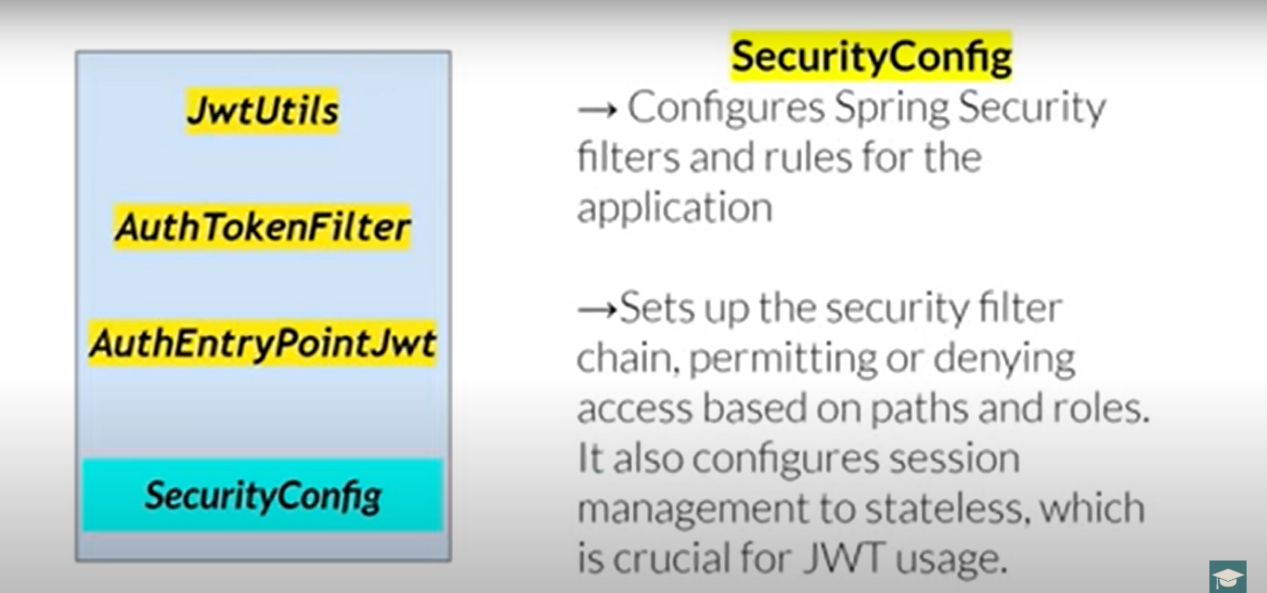












============= ===== Facing Issue while doing Coding ==================

Main ga Table scripet ni “schema.sql “ use chesi run chesinappudu “CommandLineRunner “ ni use chesanu and H2 in-memory data base ni use chesanu

Issue : tables anevi create avvayi with data tho patu but postman lo user credentials ecchi URL ni hit chesinappudu “Inavalid username or password “ ani nenu rasina code lo ni logg error vachhindi but ecchina crdential correct but JWT token anedi Genarate avvaledu.. ?

ANS : CommandLineRunner ni vadi users ni DB lo insert chese code lo small mistakes vunnay

Alright! Let's break it down in a simple way:

### 1. ****What is**** UserDetailsService****?****

UserDetailsService is an interface in Spring Security that helps to **fetch user details** (like username, password, and roles) from a database or any other source.

Think of it like a **"user database connector"** that Spring Security calls when someone tries to log in.

### 2. ****What is**** JdbcUserDetailsManager****?****

JdbcUserDetailsManager is a class that **implements** UserDetailsService and specifically **fetches user details from a database using JDBC** (SQL queries).

**Simply put:**

* It helps store, retrieve, and manage user information in the database.
* It has built-in methods to **create users, update users, delete users, and check if a user exists.**

### 3. ****Difference Between Your Two Methods****

Now, let’s compare your two methods:

#### ✅ ****First Method (Working Fine)****

@Bean

public CommandLineRunner initData (UserDetailsService userDetailsService) {

return args -> {

JdbcUserDetailsManager manager = (JdbcUserDetailsManager) userDetailsService;

UserDetails user = User.withUsername("user")

.password(passwordEncoder().encode("password"))

.roles("USER")

.build();

UserDetails admin = User.withUsername("admin")

.password(passwordEncoder().encode("password"))

.roles("ADMIN")

.build();

manager.createUser(user);

manager.createUser(admin);

};

}

### ✅ Why does this work?

* It **directly uses** userDetailsService, which is already managed by Spring Security.
* It assumes that the **database schema (tables) already exists**, so it does not check for the USERS table.
* The manager.createUser(user) method **registers users correctly** because it's connected to the right database.

#### ❌ ****Second Method (Throws "Invalid Username or Password")****

@Bean

public CommandLineRunner insertDefaultUsers(DataSource dataSource) {

return args -> {

JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);

// Ensure table exists by verifying database metadata

try (Connection conn = dataSource.getConnection()) {

DatabaseMetaData metaData = conn.getMetaData();

ResultSet tables = metaData.getTables(null, null, "USERS", null);

if (!tables.next()) {

System.out.println("USERS table does not exist — schema.sql may not have executed.");

return;

}

}

if (!jdbcUserDetailsManager.userExists("user")) {

UserDetails user1 = User.withUsername("user")

.password(passwordEncoder().encode("password"))

.roles("USER")

.build();

UserDetails user2 = User.withUsername("admin")

.password(passwordEncoder().encode("passwordadmin"))

.roles("ADMIN")

.build();

jdbcUserDetailsManager.createUser(user1);

jdbcUserDetailsManager.createUser(user2);

System.out.println("Users inserted successfully!");

}

};

}

### ❌ Why does this fail?

**It creates a new** JdbcUserDetailsManager **manually with** new JdbcUserDetailsManager(dataSource)**.**

* + The issue here is that this manually created instance **may not be recognized by Spring Security**.
  + So, when Spring Security tries to fetch users from the database during login, it **doesn’t use this manually created instance** and fails.

**Potential Schema Issues**

* + You added a check for the USERS table:

DatabaseMetaData metaData = conn.getMetaData();

ResultSet tables = metaData.getTables(null, null, "USERS", null);

* + If schema.sql didn’t execute properly, this check **may prevent user creation**.

**Spring Security Might Be Looking at a Different Database Source**

* + Your JdbcUserDetailsManager instance **uses** dataSource **manually**, but **Spring Security may be using a different** DataSource.
  + So, when trying to log in, **Spring Security cannot find the stored users**.

### ✅ ****How to Fix This?****

Instead of manually creating JdbcUserDetailsManager, **use the one managed by Spring Security** like in the first method:

🔹 **Solution:** Modify your second method like this:

@Bean

public CommandLineRunner insertDefaultUsers(UserDetailsService userDetailsService) { // Use the Spring-managed UserDetailsService

return args -> {

JdbcUserDetailsManager jdbcUserDetailsManager = (JdbcUserDetailsManager) userDetailsService; // Get managed instance

if (!jdbcUserDetailsManager.userExists("user")) {

UserDetails user1 = User.withUsername("user")

.password(passwordEncoder().encode("password"))

.roles("USER")

.build();

UserDetails user2 = User.withUsername("admin")

.password(passwordEncoder().encode("passwordadmin"))

.roles("ADMIN")

.build();

jdbcUserDetailsManager.createUser(user1);

jdbcUserDetailsManager.createUser(user2);

System.out.println("Users inserted successfully!");

}

};

}

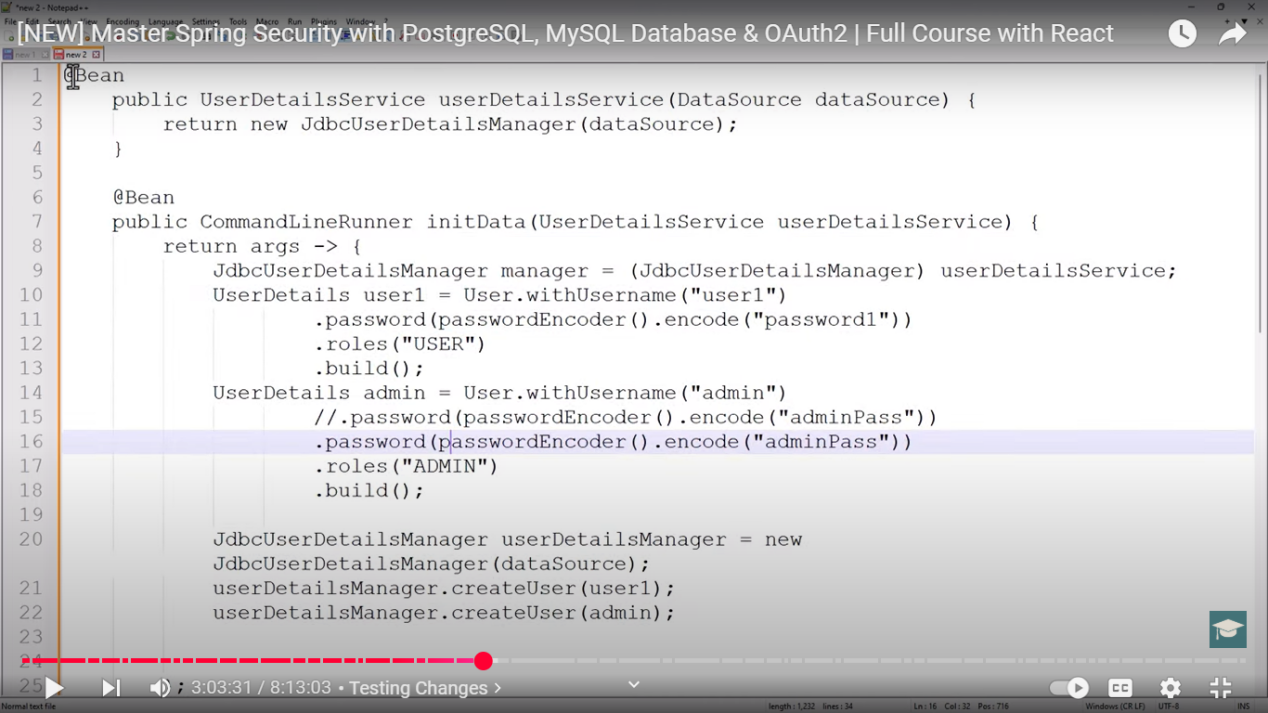
### 🎯 ****Final Key Takeaways****

✅ **Use** UserDetailsService **provided by Spring Security** instead of manually creating JdbcUserDetailsManager.  
✅ **Manually creating** JdbcUserDetailsManager **might not work correctly** because Spring Security won’t recognize it.  
✅ **Ensure your schema (tables) are properly created before inserting users.**  
✅ **Avoid checking for tables manually inside the runner**—Spring Boot handles schema creation automatically if configured correctly.

### 🚀 ****What’s Next?****

Since you're new to Spring Security, I recommend practicing:  
1️⃣ **Try using an in-memory user store (**InMemoryUserDetailsManager**)**  
2️⃣ **Experiment with different authentication providers**  
3️⃣ **Slowly build up to database authentication (**JdbcUserDetailsManager**)**

Below SS loni CommandLineRunner code ni use chesaka JWT token anedi genarate avvindi…



------------------------------- same JWT token generation with my sql --------------------------

H2 data base kakunda my sql vadiappudu…. Table script ni “schema.sql “ lo echhina sare table anedi create avvaledu but “DataSourceConfig “class ni add chesaka tables yenduku create avayi?

Great question! Let's break it down **step by step** so you can fully understand why **H2 worked automatically**, but **MySQL needed the** DataSourceConfig **class** to execute schema.sql.

## ****1️⃣ How H2 Handles**** schema.sql ****Automatically****

H2 is an **in-memory database**, and **Spring Boot has built-in support** to automatically create tables using schema.sql when you run your app.

### ****How does this happen?****

Spring Boot **automatically** executes schema.sql for **H2** because:

1. H2 is an **embedded database** (meaning it runs in-memory).
2. **Spring Boot Auto Configuration** detects that an embedded DB is used.
3. Spring Boot **automatically runs** schema.sql during startup **without any extra configuration**.

✅ **So, in H2, you don’t need** DataSourceConfig**, because Spring Boot runs** schema.sql **automatically**.

## ****2️⃣ Why MySQL Did Not Automatically Execute**** schema.sql

When you switch to **MySQL**, Spring Boot does **not** automatically execute schema.sql.

### ****Why?****

1. MySQL is **not an embedded database**—it is an **external database** that **doesn’t reset** every time your app restarts.
2. **Spring Boot does NOT run** schema.sql **automatically** for external databases like **MySQL, PostgreSQL, or Oracle** (unlike H2).
3. Spring Boot expects you to handle **database initialization manually** in MySQL.

🚨 **So, MySQL did NOT create tables automatically because Spring Boot didn’t run** schema.sql**.**

## ****3️⃣ Why Adding**** DataSourceConfig ****Fixed the Issue****

Since MySQL **does not execute** schema.sql **automatically**, you had to **force it to run manually**.

Your DataSourceConfig class does exactly that:

### ****What This Class Does****

* **Loads** schema.sql **manually** using ResourceDatabasePopulator.
* **Executes it at application startup** using DataSourceInitializer.

### ****How it Works****

@Configuration

public class DataSourceConfig {

@Autowired

private DataSource dataSource; // Injects the configured MySQL DataSource

@Bean

public DataSourceInitializer dataSourceInitializer() {

DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();

dataSourceInitializer.setDataSource(dataSource); // Connects to MySQL

ResourceDatabasePopulator resourceDatabasePopulator = new ResourceDatabasePopulator();

resourceDatabasePopulator.addScript(new ClassPathResource("schema.sql")); // Loads schema.sql

dataSourceInitializer.setDatabasePopulator(resourceDatabasePopulator);

return dataSourceInitializer; // Runs schema.sql manually

}

}

✅ **Now, MySQL runs** schema.sql**, tables get created, and everything works fine!**

## ****4️⃣ Summary (Easy Explanation)****

| **Database** | **Does Spring Boot Run** schema.sql **Automatically?** | **Why?** | **Fix Needed?** |
| --- | --- | --- | --- |
| **H2** | ✅ Yes | H2 is an embedded DB | ❌ No Fix Needed |
| **MySQL** | ❌ No | External DB, not auto-executed | ✅ Use DataSourceConfig |

## ****5️⃣ Alternative Fix (Without**** DataSourceConfig****)****

If you don’t want to use DataSourceConfig, you can **enable database initialization** in application.properties:

spring.sql.init.mode=always

This tells Spring Boot to **always** execute schema.sql, even for **MySQL**.

## ****✅ Final Takeaway****

* **H2 runs** schema.sql **automatically** because it's an **embedded** database.
* **MySQL does NOT run** schema.sql **automatically** because it's an **external** database.
* DataSourceConfig **manually forces MySQL to execute** schema.sql.
* **Alternative Fix:** Use spring.sql.init.mode=always in application.properties.

And another 2 Alternative Options yenti ante Mysql lo required Tables ni munduganee create chesukunte

schema.sql ni add cheyalsina avasaram ledu and alane DataSourceConfig class ni rayalsina avasaram ledu

Next Option Kavalsina tables ki Entity class ni create chesthe saripothundi