Downloaded Spring Project template from start.spring.io  
Spring web, spring data JPA, MySql driver, JDBC API, Lombok

Opened in Eclipse

In MySQL:

create database hotel;

DB will be created.(No table created yet, )

In Eclipse:

application.properties

Added driver, MySQL endpoint with created DB name, username, password, port.

Create few packages.

Constants, Dao, JWT, POJO(model), Rest, RestImpl, Service, ServiceImpl, Utils, Wrapper

Let’s trace the journey of a new user through your Spring Boot application, step by step. Think of it like a relay race, where each layer passes the baton forward until the registration is complete.

### ⚙️ ****Config & Security Layer****

**Where:** com.hotel.config **Includes:** SecurityConfig.java **Purpose:**

* Houses configurations like Spring Security setup, CORS policies, etc.

### 🧰 ****Utility Layer****

**Where:** com.hotel.utils **Includes:** HotelUtils.java **Purpose:**

* Provides helper methods that can be reused across different layers.

### 📜 ****Constants & Wrapper Layers****

**Where:** com.hotel.constants, com.hotel.wrapper **Purpose:**

* Constants: Stores static values (e.g., messages, error codes).
* Wrapper: Often used for custom response objects or data transfer wrappers (this one’s empty for now!).

### 🔨 0****. Application Entry Point****

**Where:** com.hotel **Includes:** HotelManagementApplication.java **Purpose:**

* Main class that launches the Spring Boot application with @SpringBootApplication.

### 🏁 1. ****Controller Layer (REST Entry Point)****

**Where:** com.hotel.rest and com.hotel.restImpl

**Includes:** UserRest.java, UserRestImpl.java **Purpose:**

* Exposes APIs to clients (e.g., frontend, Postman, mobile apps).
* Accepts HTTP requests and returns HTTP responses.
* **Communicates with the Service layer to get data.**

**File:** UserRestImpl.java

* The frontend or client sends a POST request (e.g., /user/signup) with user data in JSON format.
* The controller receives this HTTP request and invokes the appropriate method in the **Service layer**.

### 🧠 2. ****Service Layer (Business Logic)****

**Where:** com.hotel.service and com.hotel.serviceImpl

**Includes:** UserService.java, UserServiceImpl.java

**Purpose:**

* Contains business logic.
* Implements the services used by the REST layer.
* **Bridges the controller and DAO layers.**

**File:** UserServiceImpl.java

* Validates incoming data (e.g., are fields null, is the email valid).
* May check whether a user with the same email already exists by calling a DAO method.
* Encrypts the password before saving.
* Constructs a User object and passes it to the DAO layer for persistence.

### 💾 3. ****DAO Layer (Database Access)****

**Where:** com.hotel.dao **Includes:** UserDao.java **Purpose:**

* Provides interfaces for CRUD operations with the database.
* **Extends Spring Data JPA repositories to interact with your** User **entity.**

**File:** UserDao.java

* This is typically an interface extending JpaRepository<User, Integer>.
* It handles the actual interaction with the database.
* save(user) converts your User object into an INSERT query and stores it in the database.

### 🧱 4. ****Model/Entity Layer (a.k.a. Entity Class or POJO Layer)****

**Where:** com.hotel.POJO **Includes:** User.java **Purpose:**

* Represents the database structure.
* Each class is mapped to a table using JPA annotations.
* **Think of these as the “shapes” of your data.**

**File:** User.java

* This is the structure used to persist and retrieve data from the database.
* Spring JPA maps the object fields to database columns and vice versa.

### 📨 5. ****Response Back to Client****

* Once stored, the service returns a response (Success or User already exists, etc.) back to the controller, which wraps it in an HTTP response and sends it back to the client.

🔁 Flow Summary:

Client ➝ Controller ➝ Service ➝ DAO ➝ Entity ➝ Database

⬅︎ ⬅︎ ⬅︎ ⬅︎

Response (e.g., Success message)

## ✅ ****1. When a User Hits**** /signup Request Path: POST /user/signup

Client sends POST /user/signup with JSON data

|

UserRestImpl.signUp(requestMap)

|

UserServiceImpl.signUp(requestMap)

|

├── validateSignUpMap(requestMap)

|

├── userDao.findByEmailId(email)

├── if user not exists:

│ ├── getUserFromMap(requestMap)

│ │ ├── set name, contact, email

│ │ └── passwordEncoder.encode(password)

│ └── userDao.save(user)

▼

Returns ResponseEntity: "Successfully Registered" or error message

1. **UserRestImpl.signUp()**
   * Triggered by the controller.
   * Calls: userService.signUp(requestMap);

**2. UserServiceImpl.signUp()**

* Validates requestMap.
* Calls userDao.findByEmailId() to check if the user exists.
* If not, it creates a new User object:
  + Uses getUserFromMap() → which:
    - Encodes the password using BCryptPasswordEncoder
    - Sets default fields like status = false, role = user
* Calls: userDao.save(user);

## 🔑 ****2. When a User Hits**** /login Request Path: POST /user/login

1. **UserRestImpl.login()**

Client sends POST /user/login with email & password

|

UserRestImpl.login(requestMap)

|

UserServiceImpl.login(requestMap)

├── authenticationManager.authenticate(token)

│ └── Triggers CustomerUserDetailsService.loadUserByUsername(email)

│ └── Finds user via userDao.findByEmailId()

│ └── Returns Spring Security UserDetails object

├── Checks if status = "true"

│ └── Calls jwtUtil.generateToken(email, role)

│ ├── createToken(): embeds claims (role, subject)

│ └── Signs and returns JWT string

▼

Returns ResponseEntity with: {"token": "<JWT>"}

* + Triggered by the controller.
  + Calls: userService.login(requestMap);

2. **UserServiceImpl.login()**

* Builds a UsernamePasswordAuthenticationToken
* Uses: authenticationManager.authenticate(token);
* Spring Security internally invokes: CustomerUserDetailsService.loadUserByUsername(email)
* This retrieves the user and wraps it as a UserDetails object
* If status is "true", it proceeds to issue a JWT:

jwtUtil.generateToken(email, role)

* This internally:
  + Calls createToken(claims, subject)
  + Builds a token with issuedAt, expiration, role claim
  + Signs using HS256 and your secret
* Sends the token back as a JSON string

## 🔐 Bonus: How the Token is Used After Login

Once the JWT is issued:

* It’s passed in future requests via the Authorization: Bearer <token> header.
* Then, JwtFilter.doFilterInternal():
  + Extracts the token
  + Uses: jwtUtil.extractUsername(token);

jwtUtil.extractAllClaims(token);

jwtUtil.validateToken(token, userDetails);

* + Authenticates and registers the user in Spring Security’s context.

**Implementing JWT**

Create classes in JWT

JwtUtil, CustomerUserDetailsService, JwtFilter, SecurityConfig

JwtUtil:  
This class serves as a **utility for generating, extracting, and validating JWTs**, mainly for authenticating users in a secure way. It's annotated with @Service, making it a Spring Bean eligible for dependency injection. It creates the token and helps extract (or “decode”) its contents.

### 🛠 When Creating the Token:

* The method generateToken(username, role) is called during **authentication** (e.g., when a user logs in successfully).
* It internally calls createToken() which:
  + Adds custom claims (like role),
  + Sets subject (usually email or username),
  + Adds expiration time,
  + And finally signs the JWT using your secret key.

At this point, the JWT is created and returned to the client (browser/mobile app/etc.).

### 🔍 When "Decrypting" or Validating the Token:

Once a token is sent back by the client (usually as a header in an API request), these methods come into play:

* extractUsername() – pulls the username from the token.
* extractExpiration() – checks when the token expires.
* extractAllClaims() – gives access to all data inside the JWT.
* validateToken() – confirms the token is still valid and was issued for the correct user.

JwtFilter:  
This is a **custom authentication filter** that intercepts every incoming HTTP request (except for explicitly excluded endpoints like login/signup) and:

* Extracts the JWT token.
* Validates it.
* Loads the user and sets the security context in Spring Security.

## 🔐 1. How JwtFilter Integrates into the Spring Security Chain

### During app startup:

* Your SecurityConfig class adds JwtFilter before the built-in UsernamePasswordAuthenticationFilter:

http.addFilterBefore(jwtFilter,UsernamePasswordAuthenticationFilter.class);

This ensures that every incoming request goes through your custom JWT validator **before** Spring tries to authenticate using a form or basic auth.

🔁 When a request hits app (e.g., /user/login):

┌───────────────────────────┐

│ Incoming HTTP Request

└────────────┬──────────────┘

▼

SecurityFilterChain (built by Spring Security)

▼

JwtFilter.doFilterInternal()

├─> Skips token check if it's login/signup

├─> Extracts JWT from Authorization header

├─> Validates token using JwtUtil

├─> Loads UserDetails from DB

└─> Sets SecurityContext if valid

▼

UsernamePasswordAuthenticationFilter (skipped)

▼

Controller Method (now has an authenticated user!)

CustomerUserDetailsService:

It is a **custom implementation of Spring Security’s** UserDetailsService **interface**. Its job is to **load user data from your database** (via UserDao) and convert it into a format (UserDetails) that Spring Security understands.

🔐 JWT Login Authentication Sequence

🧠 Role of CustomerUserDetailsService Visually

┌───────────────────────────────┐

│ Used anytime Spring Security needs user │

│ details based on email or username │

│ │

│ Acts as a bridge: │

│ DB (UserDao) <──> Spring Security │

└──────────────────────────────────┘

▲── Used in:

- Login (during authentication)

- JwtFilter (token validation and context setup)

┌───────────────────┐

│ Client sends │

│ POST /user/login │

│ with credentials │

└────────┬───────────────┘

▼

┌───────────────────────────┐

│ UserRestImpl.login(requestMap) │

│ └── delegates to UserServiceImpl │

└────────┬──────────────────┘

▼

┌──────────────────────────────────┐

│ UserServiceImpl.login(...) │

│ └── builds UsernamePasswordAuthToken │

│ └── calls authenticationManager.authenticate(...)

└────────┬─────────────────────────┘

▼

┌────────────────────────────────────────────────────┐

│ Spring internally invokes: │

│ CustomerUserDetailsService.loadUserByUsername(email) │

│ └── Fetches User using userDao.findByEmailId() │

│ └── Wraps it as Spring Security UserDetails │

└────────┬───────────────────────────────────────────┘

▼

┌─────────────────────────────────────┐

│ AuthenticationManager returns │

│ Authentication object if success │

└────────┬────────────────────────────┘

▼

┌─────────────────────────────────────────────────────┐

│ UserServiceImpl checks userDetail.getStatus() │

│ └── If true: generates JWT via jwtUtil │

│ and sends back token to client │

└─────────────────────────────────────────────────────┘

SecurityConfig:  
This file is the **central configuration hub for Spring Security** in your application. It defines how requests are authenticated, which endpoints are public, how JWTs are processed, and how users are authenticated.

🧭 Summary Flow

App Starts

↓

SecurityConfig builds filter chain

↓

JwtFilter intercepts requests

↓

Validates JWT, sets user context

↓

Controller receives authenticated request

↓

@PreAuthorize or role checks (optional)

**Implementing Mailinator on update**

The UserWrapper.java class exists as a **DTO (Data Transfer Object)** to cleanly expose only the fields you want to **send back to clients** — without exposing sensitive or unnecessary data like passwords or roles.

**Update Operation:**

**Rest → RestImpl → Service → ServiceImpl → UserDao → User → DB → (Optional: Utility like EmailUtils)**

1. **Client sends a POST request to** /user/update

#### 2. UserRestImpl.update(requestMap)

* Acts as the HTTP endpoint entry point.
* Passes the request map to: userService.update(requestMap);

#### 3. UserServiceImpl.update(requestMap) Here's where business logic kicks in:

* **Admin Check to Ensures only admins trigger the update.:** if (jwtFilter.isAdmin())
* Fetches the User: Optional<User> optional = userDao.findById(id)
* Updates the Status: userDao.updateStatus(status, id)

update User u set u.status=:status where u.id=:id

* Sends Notifications:

sendMailToAllAdmin(status, optional.get().getEmail(), userDao.getAllAdmin())

#### 4. sendMailToAllAdmin(…) This helper method prepares email content:

* First, it removes the current admin (the one performing the action) from the getAllAdmin() recipient list.
* Then it uses your utility: emailUtils.sendSimpleMessage( . Approved/Denied . )

#### 5. EmailUtils.sendSimpleMessage(…) This handles the actual email delivery:

* Constructs a SimpleMailMessage with: from, to, cc (other admins), subject, body (includes which user was approved or disabled and by whom)
* Uses the injected JavaMailSender to dispatch the email: eMailSender.send(message)

### 📦 Data Touchpoints Along the Way

* requestMap provides the id and status
* UserDao touches the DB: fetches the user and updates the row
* JwtFilter evaluates whether the caller is an admin and tracks their email
* EmailUtils crafts and sends the outbound email
* Everything is contained within Spring’s DI system, so beans like JwtFilter, JavaMailSender, and UserDao are auto-wired where needed

**File-wise, class-wise, and operation-wise merged control and data flow**

Client (Postman / Frontend) ← HTTP POST /user/update {id, status}

|

UserRest.java ← Defines update(...) method (interface)

|

UserRestImpl.java ← Implements REST endpoint

|

UserService.java ← Declares service method update(...)

|

UserServiceImpl.java ← Business logic:

• jwtFilter.isAdmin()

• userDao.findById(id)

• userDao.updateStatus(...)

• userDao.getAllAdmin()

• jwtFilter.getCurrentUser()

• EmailUtils.sendSimpleMessage(...)

• builds and returns UserWrapper

|

├── JwtFilter.java ← Extracts and validates JWT

| → checks role

| → gets current user (admin)

|

├── UserDao.java ← DAO layer:

| → findById(id)

| → updateStatus via NamedQuery

| → getAllAdmin via NamedQuery

|

├── User.java ← JPA Entity with @NamedQuery:

| → User.updateStatus

| → User.getAllAdmin

|

├── EmailUtils.java ← Constructs and sends emails

| → uses SimpleMailMessage

|

└── JavaMailSender (Spring Boot) ← Sends the actual email

|

ResponseEntity<UserWrapper> ← Final response sent back