**Full Stack Ecommerce Website Development Using React, Spring Boot, MySql And Payment Gatway**

**Technologies Used:** **React, Spring Boot, MySQL, And Payment Gateway (RazorPay)**

**frontEnd : react , Backend : SpringBoot , Payment gateway : Razor Pay**

**Customer**

* Carousel
* sorting feature,
* pagination,
* filter,
* add to cart,
* buy,
* payment gateway,
* track order,
* review and rate product

**Admin**

* admin panel,
* add product,
* manage order,
* change order status

**Create React App Using :** *npx create-react-app project\_name or we can use npx create-react-app . (. Is used to current folder makes our project ) if we want to create new projet folder we can use npx create-react-app project\_name in this format*

**Downloads :**

**react alice carousel (2.7.1)**

**Mui :** *npm install @mui/material @emotion/react @emotion/styled*

*npm install @mdi/js @mui/material @mui/icons-material mdi-material-ui*

*npm install react-apexcharts apexcharts*

*npm install redux-thunk*

**Mui – icon :** *npm install @mui/icons-material @mui/material @emotion/styled @emotion/react*

**Tailwand css :**

* + - *npm install -D tailwindcss@3*
    - *npx tailwindcss init :* ***Created Tailwind CSS config file: tailwind.config.js***

replace **tailwind.config.js** file with below code

*/\*\* @type {import('tailwindcss').Config} \*/*

*module.exports = {*

*content: [*

*"./src/\*\*/\*.{js,jsx,ts,tsx}",*

*],*

*theme: {*

*extend: {},*

*},*

*plugins: [],*

*}*

**index.css**

Add below code In **index.css** file

*@tailwind base;*

*@tailwind components;*

*@tailwind utilities;*

**Run Project Using :** *npm start*

*----------------------------- 30/4/25 For Navigation Bar -----------------------------*

* **className (from Tailwind or CSS)**
* **sx Prop (from Material UI)**

**\src\customer\components\navigation\Navigation.jsx**

**Navigation.jsx**

**Navigation Bar :** [*https://tailwindcss.com/plus/ui-blocks/ecommerce/components/store-navigation*](https://tailwindcss.com/plus/ui-blocks/ecommerce/components/store-navigation)

(Use This link To get readyMade Navbar From Tailwind Doc )

### *npm install @heroicons/react :* ( Icons for UI This library provides SVG-based icons (like shopping carts, chevrons, user profiles, etc.) that you can use directly in React components.)

### *npm install @headlessui/react :* ***(***Accessible UI components (without styles)

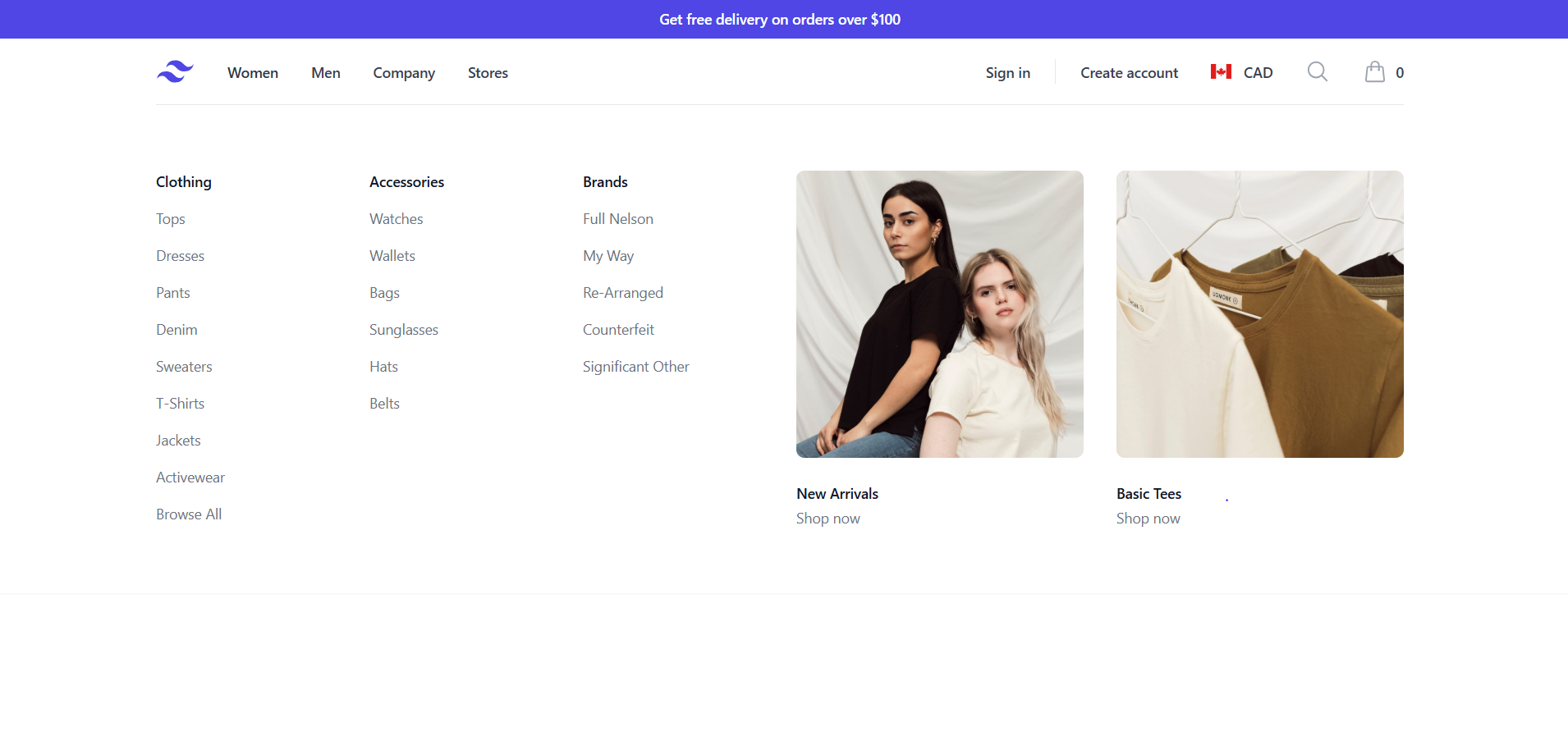
This library gives you interactive components like:

* Dialogs (modals)
* Dropdowns
* Tabs
* Menus
* Disclosures (accordions)

Install : *npm install react-router-dom react-redux*

But without any built-in styling, you can fully customize them (usually with Tailwind CSS).

So Navbar look like this



*react-alice-carousel* : is a React carousel/slider library used to create image sliders, carousels, or galleries in React applications.

**Key Features :**

Responsive — works across devices and screen sizes

* Touch-friendly — supports swipe/drag gestures on mobile
* Autoplay support
* Infinite loop
* Custom buttons/dots
* Lightweight and easy to integrate

*-------------------------------- To Create React Alice Carousel -----------------------------------*

Install : *npm i react-alice-carousel*

Go to the <https://maxmarinich.github.io/react-alice-carousel/> and copy the code of Basic Example

And paste it into **\src\customer\components \ HomeCarousel \ MainCarousel.jsx** and then use < MainCarousel > inside **\src\customer\components \pages\HomePage.jsx**

**Images are stored inside \src\customer\components \ HomeCarousel \ MainCarouselData.js**

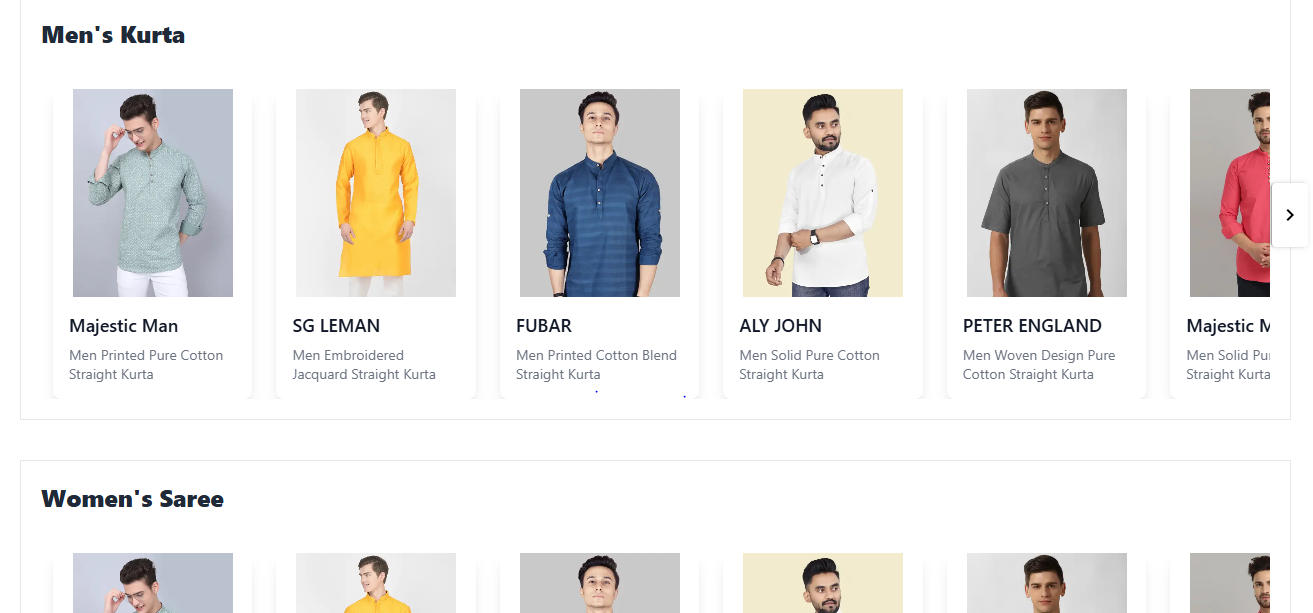


*-------------------------- To Create HomeSection Card --------------------------------*

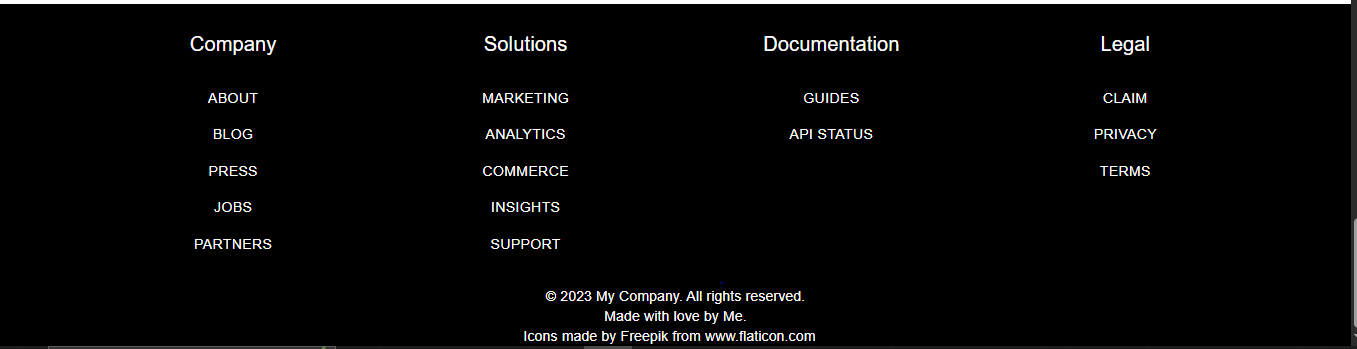
Created **HomeSectionCard** inside **\src\customer\components \ HomeSectionCard \ HomeSectionCard.jsx** created to display images in card format

Created **HomeSectionCard** inside **\src\customer\components \ HomeSectionCarousel**

**\ HomeSectionCarousel.jsx** created to display image card’s in responsive manner **<HomeSectionCard/>** is used inside it



Created **Footer.jsx** inside **\src\customer\components \ Footer\ Footer.jsx**



*---------------------------------- 01/05/2025 Product Page ----------------------------------------*

For filters: <https://tailwindcss.com/plus/ui-blocks/ecommerce/components/category-filters>

For Radio Button: <https://mui.com/material-ui/react-radio-button/>

For filtericon : https://mui.com/material-ui/material-icons/?query=filter&selected=FilterList

Product component Is created inside it 3 files are created

**\src\customer\components \ Product\ Product.jsx**

* Used to filter and add dynamically product data which is displayed on screen

**\src\customer\components \ Product \ ProductCard.jsx**

* Card is designed (rg. Its size h,w etc)

**\src\customer\components \ Product \ ProductCard.css**

* Css for card , applied animation, transformation
* For location we used **react-router-dom**

The react-router-dom library is used in React applications to **handle routing**, i.e., navigation between different pages or components **without reloading the entire web page**. It allows you to create **single-page applications (SPAs)** with **multiple views**.

**Single Page Application (SPA):**

* Only parts of the UI change when the route changes; the page doesn’t reload completely.

**Navigation Components:**

* <Link> and <NavLink>: Used instead of <a> tags for internal navigation.
* <Navigate>: Redirects users programmatically.

**Route Matching & Rendering:**

* <Routes> and <Route> decide which component to render based on the URL.

---------------------------------------- 02-05-2025 Product Data -------------------------------------

For Rating : <https://mui.com/material-ui/react-rating/>

Form Progress: https://mui.com/material-ui/react-stepper/

**\src\customer\components \ ProductDetails\ ProductDetails.jsx**

**ProductDetails.jsx –** It contains all details about the product like it’s size,price, rating,review etc.

**ProductDetailsCard.jsx-** It contains all ratings and reviews which is imported inside the **ProductDetails.jsx**

Inside **ProductDetails.jsx** different sections are created for rating and reviews, similar products

**\src\customer\components \ Cart\ Cart.jsx –** item details are displayed like price, discount, delivery charges

**\src\customer\components \ Cart \ CartItem.jsx-** cart is imported inside it

//--------------------------

**\src\customer\components \ Checkout \ Checkout.jsx**

**\src\customer\components \ Checkout \ DeliveryAddressForm.jsx**

* AddressCard.jsx is imported in DeliveryAddressForm.jsx

**\src\customer\components \ Checkout \ OrderSummary.jsx**

**\src\customer\components \ AddressCard \ AddressCard.jsx**

*--------------------------03/05/2025------------------------*

**Flow**

**HomePage ->(nav -> women->top)->Product Page(by clicking on any product)-> Product Detail (Add To Cart) -> Cart (ckeckout) -> Delivery Details**

**My Orders -> Order.jsx**

**//=========================== Backend =======================**

**SpringBoot**

**Spring Security**

**Jwt-authentication**

**Signup/Signin - API**

*--------------------------05/05/2025------------------------*

*Created com.arc.config package*

* *AppConfig*

This is a Spring Boot **security configuration class** that:

* Secures API endpoints using **JWT**
* Configures **stateless** sessions
* Sets up **CORS** for frontend apps
* Enables **password encryption**
* Allows both **HTTP Basic** and **Form Login**

### securityFilterChain(HttpSecurity http)

**1. Stateless Session**

***http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)***

No session will be created; every request must carry authentication (like a JWT).

### SessionCreationPolicy.STATELESS

* **No HTTP session is created or used.**
* Every request is **independent** and must carry full authentication (e.g., JWT token).
* Commonly used in **REST APIs**, where you want scalability and stateless communication.
* No HttpSession object on the server.

#### Use case:

JWT-based APIs, microservices.

### SessionCreationPolicy.STATEFUL

Actually, the equivalent in Spring Security is:

***SessionCreationPolicy.IF\_REQUIRED***

But if you don’t set anything, Spring Security uses stateful sessions **by default**.

#### When using session-based (stateful) auth:

* Spring will create a HttpSession after the user logs in.
* Session stores the user's authentication details.
* Future requests reuse the same session, without re-authenticating.
* Server memory is used to store sessions.

#### Use case:Traditional login apps (e.g., Spring MVC + JSP/Thymeleaf), where users maintain a session.

|  |  |  |
| --- | --- | --- |
| **Policy** | **Behavior** | **Use Case** |
| STATELESS | No session; every request must include authentication | APIs with JWT |
| IF\_REQUIRED (default) | Session created **only if needed** | Standard web apps |
| ALWAYS | Always create a session | Not commonly used |
| NEVER | Never create a session, but use existing if present | Rare |

**2. Authorization Rules**

***.authorizeHttpRequests(auth -> auth***

***.requestMatchers("/api/\*\*").authenticated()***

***.anyRequest().permitAll())***

Any request to /api/\*\* must be **authenticated**.  
Other routes are **open to everyone**.

3. **JWT Filter**

***.addFilterBefore(new JwtValidator(), BasicAuthenticationFilter.class)***

Adds a **custom JWT validator** filter **before** Spring's built-in filter to check JWT tokens.

4. **CORS Configuration**

**.*cors().configurationSource(...)***

Allows frontend apps on http://localhost:3000 and :4200 (React/Angular) to access the backend.  
Allows **all HTTP methods and headers**, exposes the Authorization header.

* **CORS** stands for **Cross-Origin Resource Sharing**.

It is a **security feature in browsers** that controls how resources (like APIs) can be **accessed from different domains**.

### Why is CORS Needed?

By default, the browser **blocks requests** made from one origin (domain/port) to another for security reasons.

#### For example:

* Your backend is running at: http://localhost:8080
* Your React frontend is running at: http://localhost:3000

When React tries to call the backend API, the browser blocks it **unless** the backend **explicitly allows it** using **CORS**.

* **How CORS Solves This**

When you enable CORS in your backend, you tell the browser:

"It’s okay to allow requests from this other origin."

So your Spring Boot backend uses:

***cfg.setAllowedOrigins(Arrays.asList(***

***"http://localhost:3000",***

***"http://localhost:4200"***

***));***

This lets your React or Angular app communicate with the backend without being blocked.

5. **CSRF Disabled**

***.csrf().disable()***

Disables CSRF protection (common for APIs using tokens instead of cookies).

6. **Enables HTTP Basic and Form Login**

***.and().httpBasic().and().formLogin()***

Enables both HTTP Basic and traditional login form authentication (not typically used with stateless APIs, but allowed here).

**passwordEncoder()**

***return new BCryptPasswordEncoder();***

Provides a **BCrypt encoder** for hashing passwords before storing them in the database (secure and recommended).

* *JwtValidator*

It ensures that **every incoming request** with a JWT token is:

* **Extracted**
* **Verified (signature, expiration, etc.)**
* **Converted into a valid Spring Authentication object** (for authorization)

**How It Works**

### 1. ****Extends**** OncePerRequestFilter

* Ensures this filter runs **once per request** (not multiple times).

2. **Reads the JWT token from header**

**String jwt = request.*getHeader(JwtConstant*.JWT\_HEADER);**

* Typically reads from: Authorization: Bearer <JWT\_TOKEN>
* Removes "Bearer " prefix using: ***jwt = jwt.substring(7);***

3. **Verifies Token Signature**

***SecretKey key = Keys.hmacShaKeyFor(JwtConstant.SECRET\_KEY.getBytes());***

***Claims claims = Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(jwt).getBody();***

 Parses and **validates the JWT** using the secret key.

 Extracts payload (claims) like email, authorities, etc.

### 4. ****Creates Spring Security Authentication****

***List<GrantedAuthority> auths = AuthorityUtils.commaSeparatedStringToAuthorityList(authorities);***

***Authentication authentication = new UsernamePasswordAuthenticationToken(email, null, auths);***

* Converts the authorities string to Spring Security roles.
* Creates an Authentication object.

uthentication should be stored in the **SecurityContext**,

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

Without this line, the user is **not treated as authenticated** in Spring Security.

### 5. ****Exception Handling****

* If the token is invalid or expired, it throws:

***throw new BadCredentialsException("Invalid token...");***

6. **Calls next filter**

***filterChain.doFilter(request, response);***

|  |  |
| --- | --- |
| **Part** | **What it does** |
| Read Header | Gets the JWT from Authorization |
| Validate | Verifies the JWT using the secret key |
| Extract | Gets email and authorities from claims |
| Convert | Builds Authentication object |
| Forward | Passes request to next filter |

* *JwtConstant*
* *JwtProvider*

The JwtProvider class is responsible for:

1. **Creating JWT tokens** when a user logs in.
2. **Extracting data (email)** from an incoming JWT.

## Explanation of Key Parts

### ***SecretKey key = Keys.hmacShaKeyFor(...)***

* Uses a **shared secret key** (from JwtConstant.SECRET\_KEY) to sign and validate tokens.
* Ensures integrity and authenticity of the JWT.

### *****generateToken(Authentication auth)*****

Creates a JWT token containing

* **issuedAt**: Current time
* **expiration**: After 24 hours (actually set to ~9.8 days because 846000000 ms ≈ 9.8 days).
* **claim**: Stores user’s email.
* **signWith(key)**: Signs the token.

**Returns** a compact JWT string to be sent to the client.

### 2. getEmailFromToken(String jwt)

* Removes "Bearer " prefix.
* Parses the JWT.
* Extracts and returns the email claim from the token.

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| generateToken(...) | Creates a signed JWT with email |
| getEmailFromToken | Reads JWT and extracts email |
| SecretKey | Ensures JWT is signed/validated |

*-------------------------------------------------------------------------------------------------------------------------*

**JWT (JSON Web Token)** is a compact, URL-safe means of representing claims between two parties. It's commonly used for **authentication and authorization** in web applications.

When a user logs in, the server creates a session and stores it on the server. However, this is **stateful** and not scalable. JWT allows **stateless** authentication.

## Structure of a JWT

A JWT has 3 parts: *Header.Payload.Signature*

 **Header**: type of the token (JWT) and signing algorithm (e.g., HMAC SHA256).

 **Payload**: data or claims (like user id, roles, expiry).

 **Signature**: to verify that the token is not altered.

## Claim in JWT

In **JWT (JSON Web Token)**, a **claim** is a piece of information about an entity (typically the user) and additional metadata. Claims are stored in the **payload** section of the token and are used to share **identity data** between two parties: the **issuer** (server) and the **consumer** (client or another server).

It Enable stateless **server-side authentication**

JWT defines **three types of claims**:

### 1. ****Registered Claims**** (Predefined by JWT spec)

**2. Public Claims**

* Custom claims that are **agreed upon by both parties**.
* Should be **namespaced** to avoid collisions.

**3. Private Claims**

* Fully custom claims used between your client and server.
* You can store anything useful like roles, email, userId.

## **Authentication**?

### ➤ ****Definition:****

**Authentication** is the process of **verifying** who someone is.

Think of it as **"proving your identity"**.

**Example:**

When you **log in** to a website using a username and password:

* The system checks whether your credentials are correct.
* If yes, you are **authenticated**.

**In a Spring Boot App:**

* You provide your credentials (POST /login).
* The app validates them (maybe via a database).
* If valid, it generates a **JWT token** or **session**.
* That token is sent in subsequent requests to identify you.

## **Authorization**?

### ****Definition:****

**Authorization** is the process of **verifying what you have access to**.

It determines **what actions** or **resources** a verified user is **allowed to access**.

### ****Example:****

You log in to a system (authenticated), but:

* Only **admin** users can access /admin/dashboard.
* If your **role is not "admin"**, you will get a **403 Forbidden**.

This is **authorization** in action.

### In a Spring Boot App:

* After authentication, the app checks your **roles or authorities**.
* Example roles: USER, ADMIN, MANAGER.
* It restricts access to URLs, methods, or actions based on roles.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Authentication** | **Authorization** |
| **Purpose** | Who are you? | What are you allowed to do? |
| **Occurs when?** | Before authorization | After authentication |
| **How?** | Username/password, token, etc. | Roles, permissions, access rules |
| **Output** | Identity is verified | Access is granted or denied |
| **Example** | Login form verification | Access control to admin panel |

### What is @JsonIgnore?

@JsonIgnore is an annotation provided by the **Jackson library** (which handles JSON serialization/deserialization in Spring Boot).

It tells Jackson to **ignore a specific field** when converting a Java object to JSON or vice versa.

Use Case

*public class User {*

*private String name;*

*@JsonIgnore*

*private String password;*

*// Getters and Setters*

*}*

JSON Output:

*{*

*"name": "Pallavi"*

*}*

## When to Use

|  |  |
| --- | --- |
| **Situation** | **Why @JsonIgnore Helps** |
| Hide sensitive fields | Like passwords or internal IDs |
| Avoid circular references | In bi-directional relationships (@OneToMany, @ManyToOne) |
| Reduce payload | Exclude unnecessary data from API responses |

@JsonIgnore only affects **JSON conversion** — not persistence (i.e., it's still saved in the database).

To use it, ensure you have **Jackson** in your dependencies (Spring Boot includes it by default).

07/05/2025

Login register api integration

**authentication part**

**login and register**

**redux**

**axios- for api dispatch**

Deployment 08/05/2025

* *Fontend : https://vercel.com/*
* *Backend:* railway.app/dashboard