i am full stack developer , i want create one product application using spring boot and angular , i need to create first spring boot app to create apis--> i need add, get , update , delete apis ---> then security should be added in --> we will handel it using jwt --> so most of the get api will open and update add delete api will be under security only admin can access them ---> so this for spring boot ---> then we are going to implement frontend in which there will be page which shows all the products ---> one home page --> at all products page by clicking on produts we can see produt details ---> then there is login page in which admin can login --> admin dashboaerd will open --> in that there will be the all products list in table format --> for each produt there is option for delete and update --> then above the all products list there should be one option for adding product

**Spring Boot Backend**

* I want to set up Spring Boot project I need it to use Maven for dependencies, and I'll definitely need Spring Web for our APIs, Spring Data JPA to talk to the database, and H2 database for development for now. name this project 'ProductService'.
* "Now that the project is created, can you help me configure the application.properties file? I want to set up the H2 database connection and also tell JPA to automatically update the database schema based on our entities. So, set spring.jpa.hibernate.ddl-auto to update."
* " create a Product entity . It should have these fields: an id (which will be a Long and the primary key, auto-generated), name (a String), description (also a String), price (a Double), and imageUrl (another String for product images)."
* “I need a way to interact with the database. generate a Spring Data JPA repository for the Product entity. name it ProductRepository."
* " create a ProductController class. This will be where we define all our product-related API endpoints."
* “add an endpoint to fetch *all* products. Define a GET endpoint at /api/products that will retrieve and return a list of all products from the database."
* "I also need a way to get a single product by its unique identifier. implement another GET endpoint at /api/products/{id} that retrieves a specific product using its ID from the path."
* “Create a POST endpoint at /api/products that allows us to add a new product. Make sure it can take a Product object in the request body, so use @RequestBody."
* " need to update product details. Implement a PUT endpoint at /api/products/{id} for updating an existing product. add logic to handle cases where the provided product ID doesn't exist; it should respond gracefully."
* “add a DELETE endpoint at /api/products/{id} to remove a product from the database by its ID."

**JWT**

* “add security. include the Spring Security and JJWT (Java JWT) dependencies in our pom.xml file."
* "For authentication, we need a User entity. Create a User entity with fields: an id (Long), a unique username (String), a password (String), and roles (a String, for example, it might hold 'ROLE\_ADMIN,ROLE\_USER')."
* "Similar to products, generate a Spring Data JPA repository for our User entity. UserRepository."
* " Create a UserService interface and its concrete implementation. This service will be responsible for handling user details and authentication logic."
* "To integrate with Spring Security, I need to implement the UserDetailsService interface. In our UserService (or a dedicated class), implement the method to load a user by their username for authentication."
* " Configure Spring Security
  + disable CSRF (Cross-Site Request Forgery) protection for our API.
  + Allow access to /api/auth/\*\* endpoints without authentication.
  + For all other requests, require authentication.
  + And importantly, set session management to stateless since we're using JWTs."
* "We need a custom filter to intercept requests and validate JWT tokens. Create a custom JWT filter class. This filter should be able to validate tokens found in incoming requests and then set the authentication context for Spring Security."
* "create a helper class for JWT operations. Generate a JWT utility class. It should contain methods to generateToken (for login) and validateToken, as well as a method to extractUsername from a token."
* "Now for the login API. Create an AuthController class. Inside it, define a /api/auth/login endpoint. This endpoint should accept a username and password, authenticate the user, and if successful, return a newly generated JWT token."
* Securing Product APIs (Admin Only): "With security in place, let's protect our product management APIs. Go back to the ProductController and secure the POST, PUT, and DELETE endpoints for /api/products. These should only be accessible by users with the 'ROLE\_ADMIN' role."
* " ensure that the GET endpoints for /api/products and /api/products/{id} remain open and accessible to *all* users, even unauthenticated ones."

**Angular Frontend**

* “generate a new Angular project using the Angular CLI. 'ProductApp'."
* “add Angular Material to our project. We'll use its components for consistency and ease of styling."
* “Configure the routing module to include routes for our main pages:
  + A home page (/).
  + A products list page (/products).
  + A single product detail page (/products/:id).
  + A login page (/login).
  + And an admin dashboard (/admin-dashboard)."
  + Generate a HomeComponent.
  + Generate a ProductListComponent (this will show all products).
  + Generate a LoginComponent.
* " make navigation easy, create a basic navigation bar in our app.component.html. It should have links to the Home, Products, and Login pages."
* “ define an interface named Product. It should have properties matching our backend entity: id, name, description, price, and imageUrl."
* "Create a ProductService that will handle all our HTTP requests to the Spring Boot /api/products endpoints. Make sure it uses Angular's HttpClient."
* "Inside our ProductService, implement a method called getAllProducts(). This method should make the HTTP request to fetch all products from our backend."
* " in ProductService, implement a method named getProductById(id: number). This will fetch a single product based on its ID."
* "In the ProductListComponent, inject the ProductService. When the component initializes (ngOnInit), call getAllProducts() to fetch the data. Then, use \*ngFor in the template to display the list of products, showing at least the product name, price, and image."
* "On the ProductListComponent, make each displayed product clickable. When a user clicks on a product, it should navigate them to the ProductDetailComponent, passing the product's ID as a route parameter."
* "In the ProductDetailComponent, retrieve the product ID from the active route. Then, use the ProductService to fetch the specific product's details based on that ID. Finally, display the product's name, description, price, and image prominently."
* " create an AuthService using HttpClient. This service will manage all our authentication logic."
* "In AuthService, implement a login(username, password) method. This method should send a POST request to /api/auth/login on our backend. If the login is successful, it needs to store the received JWT token in localStorage."
* "Add additional methods to AuthService:
  + One to checkIfUserIsLoggedIn() (by checking for the JWT in localStorage).
  + Another to retrieveJwtToken().
  + And a logout() method to clear the JWT token from localStorage."
* "Design the LoginComponent with simple input fields for username and password, and a prominent login button."
* "When the login form in LoginComponent is submitted, call AuthService.login(). If the login is successful, redirect the user to the /admin-dashboard. If there's an error, display a clear, user-friendly error message."
* "To protect our admin section, generate an Angular Guard named AdminGuard. Implement its canActivate method. This guard should check if the user is authenticated (using AuthService) and specifically if they have the 'ROLE\_ADMIN' role (you'll need to decode the JWT or check a role claim from the backend). If they are not authorized, redirect them to the login page."
* " apply the AdminGuard to the /admin-dashboard route in our Angular routing module, ensuring only authorized users can access it."
* "AdminDashboardComponent, display a comprehensive list of all products. Use an Angular Material table for this. Each row should clearly show product details."
* "For each product row in the table, add two action buttons: an 'Edit' button and a 'Delete' button."
* "Above the product table, add a prominent 'Add Product' button."
* "Go back to our ProductService. Implement the addProduct(product: Product) method. This method must send a POST request to /api/products and, critically, include the JWT token in the Authorization header."
* "Also in ProductService, implement the updateProduct(id: number, product: Product) method. This will send a PUT request to /api/products/{id}, and it *must* also include the JWT token in the Authorization header."
* "And for deletion, implement the deleteProduct(id: number) method in ProductService. This will send a DELETE request to /api/products/{id}, again, including the JWT token."
* "To simplify security, create an HttpInterceptor. This interceptor should automatically add the JWT token to the Authorization header of *all* outgoing HTTP requests, except for the login endpoint itself."
* "Back in AdminDashboardComponent, use the ProductService to fetch and populate the product table with all products upon initialization."
* "Implement the logic for the 'Delete' button. When clicked, prompt the user for confirmation, then call ProductService.deleteProduct(). After a successful deletion, refresh the product list in the table."
* "Implement the 'Add Product' button's functionality. When clicked, it should navigate the user to a dedicated ProductFormComponent (or open it in a dialog, whichever is simpler for now)."
* "Implement the 'Edit' button's functionality. When clicked, it should navigate the user to the ProductFormComponent, but this time, it needs to pass the product ID so the form can pre-populate with existing details for editing."
* "Generate a ProductFormComponent. Design a user-friendly form within it that includes input fields for the product name, description, price, and imageUrl."
* "If the ProductFormComponent is being used for an *edit* operation, ensure it retrieves the product ID from the route and pre-populates the form with the existing product's data."
* "On form submission in ProductFormComponent, call either ProductService.addProduct() or ProductService.updateProduct(). The choice depends on whether it's a new product or an edit. After a successful operation, navigate the user back to the AdminDashboardComponent."
* "Implement basic error handling in our component methods that make HTTP requests. When an API call fails, display user-friendly error messages to the user (e.g., using Angular Material's SnackBar component)."