 git config --global user.email "shubhendusaurav45@gmail.com"

 git config --global user.name "shubhendusaurav"

-- Create users with passwords

CREATE ROLE product WITH LOGIN PASSWORD 'product';   
CREATE ROLE order\_user WITH LOGIN PASSWORD 'order';   
CREATE ROLE user\_db WITH LOGIN PASSWORD 'user';

-- Create databases owned by respective users

CREATE DATABASE productdb OWNER product;   
CREATE DATABASE orderdb OWNER order\_user;   
CREATE DATABASE userdb OWNER user\_db;

Study notes-  
  
Starting from Jakarta EE 9, the namespace for all Jakarta EE specifications (including Servlets) was changed from javax.\* to jakarta.\*. Since you're using Spring Boot 3.3.2 and Java 17, it's likely that you're on Spring Framework 6.x and above, which uses the jakarta namespace.

User-service-

**/authenticate**:

* This endpoint is used to authenticate users and provide a JWT token. It involves checking the provided credentials and issuing a token if valid. This process does not require a JWT token initially.
* The security configuration should allow access to this endpoint without a token.

**Other APIs**:

* These endpoints are protected by the JWT token mechanism. The JwtAuthenticationFilter intercepts requests to these endpoints, checks for the presence and validity of a JWT token, and sets the authentication context accordingly.
* If the token is absent or invalid, access is denied, and appropriate responses are returned (e.g., HTTP 401 Unauthorized).

To understand how a JWT (JSON Web Token) issued by the User Service is validated in the Product Service, it's crucial to delve into the JWT structure, the role of the JWT decoder, and the Spring Security configuration in the Product Service. Here’s a detailed explanation of the workflow:

**JWT Structure**

A typical JWT consists of three parts:

1. **Header**: Contains metadata about the token, including the type of token and the signing algorithm.
2. **Payload**: Contains the claims, which are statements about an entity (typically, the user) and additional data. This can include standard claims (like sub for subject, exp for expiration) and custom claims.
3. **Signature**: A cryptographic signature used to verify the token's integrity and authenticity.

A JWT is encoded as a Base64Url string: header.payload.signature.

**JWT Token Issuance**

1. **User Authentication**:
   * A user authenticates with the User Service by providing credentials (e.g., username and password).
   * The User Service verifies these credentials.
2. **JWT Creation**:
   * Upon successful authentication, the User Service generates a JWT containing the user's details and roles.
   * **The JWT is signed using a private key (asymmetric) or a secret key (symmetric).**
3. **JWT Delivery**:
   * The User Service sends the JWT to the client, typically in the response body or headers.

**JWT Validation in Product Service**

**1. Receiving the Token**

* **Client Request**: The client includes the JWT in the Authorization header of the HTTP request to the Product Service. The header looks like: Authorization: Bearer <jwt\_token>.

**2. Token Extraction and Parsing**

* **Bearer Token Extraction**: Spring Security's filter chain intercepts the incoming request. The JWT is extracted from the Authorization header.

**3. Token Validation Workflow**

**a. Signature Verification**

* **Decoding the Token**: The JwtDecoder configured in the Product Service decodes the token.
* **Signature Verification**: The decoder verifies the signature using:
  + **Public Key (Asymmetric Signing)**: If the JWT was signed using a private key, the public key (retrieved from a JWKS endpoint or a trusted source) is used to verify the signature.
  + **Secret Key (Symmetric Signing)**: If the JWT was signed with a secret key, the same key is used for verification.

This step ensures that the token has not been tampered with and was indeed issued by a trusted entity (User Service).

**b. Claims Verification**

* **Expiration (exp)**: The exp claim is checked to ensure the token is still valid and hasn't expired.
* **Audience (aud) and Issuer (iss) Claims**: These are optionally checked to ensure the token is intended for the Product Service and issued by the correct authority.

**c. Authentication Context Setup**

* **Extracting User Details**: Once the token is verified, the claims in the JWT (such as user roles and permissions) are used to create an Authentication object.
* **Setting Security Context**: The Authentication object is set in the SecurityContext, making the user details available for the duration of the request. This allows the Product Service to authorize actions based on the user's roles and permissions.

**4. Authorization**

* **Access Control**: Based on the roles and permissions extracted from the JWT, the Product Service decides whether the authenticated user is allowed to access the requested resource (e.g., /products/\*\* endpoint).

**5. Response**

* If the token is valid and the user is authorized, the Product Service processes the request and returns the appropriate response.
* If the token is invalid, expired, or the user is not authorized, an error response (such as 401 Unauthorized or 403 Forbidden) is returned.

**Summary**

The Product Service relies on the JwtDecoder to handle the complex process of validating the JWT. This includes verifying the token's signature, checking claims, and setting up the security context. By delegating these tasks to the JWT infrastructure provided by Spring Security, the Product Service can focus on its core functionality, trusting the security infrastructure to handle authentication and authorization based on the verified token.

**Validation Steps in Product Service**

1. **Token Extraction**: The JWT is extracted from the Authorization header.
2. **Signature Verification**:
   * The JwtDecoder verifies the JWT's signature using the provided secret key. If the signature is invalid, the token is rejected.
3. **Claims Verification**:
   * Standard claims like exp (expiration) and iat (issued at) are checked to ensure the token is valid.
   * Additional custom claims can also be verified as needed.
4. **Authentication Context**:
   * If the token is valid, Spring Security creates an Authentication object with details from the token (e.g., username, roles).
   * This Authentication object is stored in the SecurityContext, making the user details accessible for authorization checks during the request.
5. **Authorization**:
   * The Product Service checks if the authenticated user has the necessary permissions to access the requested resources (e.g., /products/\*\*).
6. **Response**:
   * If the user is authorized, the Product Service processes the request and returns the response.
   * If the token is invalid or the user is not authorized, an appropriate error response (e.g., 401 Unauthorized) is returned.