What Happens When a Request Comes in?

Step-by-Step Explanation (Starting from a Login Request):

1. Client Makes a Request to Login API

When the client (e.g., a user in Postman or your browser) makes a **POST request** to the /auth/login endpoint, it will send the **username** and **password** in the request body.
For example, the request body could look like this:

```
{
  "username": "john_doe",
  "password": "password123"
}
```

2. AuthController Handles the Request

The request is received by the **AuthController.java** class. In this class, you have an endpoint mapped for login (and registration).

What happens inside AuthController?

- The AuthController receives the POST /auth/login request.
- It expects a **AuthRequest** object, which contains the username and password (this is the **DTO** — Data Transfer Object).

Now, the AuthController.java will delegate the logic to the **UserDetailsServiceImpl** to check if the user exists and validate the credentials.

3. AuthRequest DTO

DTO (Data Transfer Object) is simply a container for data that we send between layers.

- **AuthRequest.java** is a **DTO** that the AuthController uses to **map** the incoming request body (username, password).
- When the AuthController receives the request, it automatically maps it to the AuthRequest DTO.

```
public class AuthRequest {
   private String username;
   private String password;

   // Getters and Setters
}
```

This DTO is sent to the **Service Layer** to perform the actual login process.

4. UserDetailsServiceImpl Loads the User

In the **UserDetailsServiceImpl.java** class:

- We implement **UserDetailsService** interface provided by Spring Security. This is how Spring Boot **loads the user** for authentication purposes.
- **loadUserByUsername** method checks if the user exists in the database. It interacts with the **UserRepository** to fetch the user.
- The UserRepository (which is a Spring Data JPA repository) queries the database to retrieve the user.

Example Flow:

- If the user exists, UserDetailsServiceImpl returns a User object, containing the **username**, **password**, and **role**.
- If the user doesn't exist, it throws an exception (typically UsernameNotFoundException).

5. Password Validation and JwtService (Token Generation)

Once the **UserDetailsServiceImpl** returns the user data, the **AuthController** now checks if the provided password matches the one stored in the database.
Here's the flow:

- **BCryptPasswordEncoder** (configured in Spring Security) is used to **validate the password** against the one stored in the database.
- If the password is valid, we move to **token generation** using the **JwtService.java**.
- The **JwtService.java** generates a JWT token based on the **username** and **role**.

The token is then returned to the user as a response.

6. Security Config - Configuring Spring Security

At this point, the user is logged in and has a JWT token. Now, Spring Security comes into play to secure protected routes (like /api/user, /api/admin).
SecurityConfig.java is where we configure the entire security setup:

- We tell Spring Security how to handle **authentication** and **authorization**.
- We configure JWT token authentication using the **JwtFilter.java** to validate tokens for each request to protected routes.

Key Points:

- **Configure HTTP Security**: Spring Security needs to know which endpoints are **public** and which are **protected**.
    - Public routes (like login and registration) should be excluded from authentication.
    - Protected routes (like /api/user, /api/admin) will require JWT token validation.
- **Configure Authentication Manager**: This is used to authenticate users based on the JWT token.

7. JWT Filter (JwtFilter.java)

This is the **security filter** that intercepts incoming requests to protected endpoints and validates the JWT token. If the token is valid, the request is allowed to proceed.

What does JwtFilter.java do?

- It reads the **Authorization header** from the request (which contains the JWT token).
- If the token is present, it **validates** the token using **JwtService**.
- If valid, it sets the **Authentication** context in Spring Security (so that Spring can handle authorization).
- If invalid, it responds with an **Unauthorized (401)** status.

8. TestController (Protected Routes)

When the user sends a request to a protected route, like /api/user or /api/admin, the **JwtFilter** is triggered to check the validity of the JWT token.

- If the JWT token is valid, Spring Security allows access to the **TestController** routes.

- These routes will check the user's role to ensure they have appropriate permissions. For example:
    - **Admin routes** (/api/admin) might require the user to have the **ROLE_ADMIN**.
    - **User routes** (/api/user) can be accessed by any authenticated user with **ROLE_USER**.

Final Flow:

1. **Client Request to /auth/login**: The request hits the **AuthController**.
2. **AuthController** calls **UserDetailsServiceImpl** to find the user.
3. **UserDetailsServiceImpl** interacts with **UserRepository** to fetch user data.
4. **Password Validation**: The controller checks the password.
5. **JWT Token Generation**: If credentials are correct, **JwtService** generates a token.
6. **SecurityConfig**: Spring Security configuration handles how routes are protected and which ones are public.
7. **JwtFilter**: Intercepts requests to check if the token is valid.
8. **TestController**: Once authenticated, protected routes (like /api/user, /api/admin) are accessible.

Recap:

- **Controller Layer**: Handles HTTP requests and calls services to perform business logic.
- **DTOs**: Used for transferring data between layers (e.g., AuthRequest, AuthResponse).
- **Service Layer**: Contains business logic, like password validation and token generation.
- **Repository Layer**: Handles database access for CRUD operations.
- **Security Config**: Configures Spring Security to secure endpoints.
- **JWT Filter**: Validates tokens for each request.

Conclusion

This explanation should help you understand how data flows from the **Controller Layer** to the **Service Layer**, **Repository Layer**, and **Security Layer** in a Spring Boot application with JWT authentication.

Each layer is **responsible for a specific concern**:

- **Controller**: Request handling.
- **Service**: Business logic.
- **Repository**: Data access.
- **Security**: Authentication and authorization.