---

1. Application Setup

Application starts from:

```
@SpringBootApplication
public class SpringSecurityJwtApplication implements
CommandLineRunner{
```

- This is the **main class**.
- When it runs, it does two things:
    1. Starts the Spring Boot app.
    2. Creates a test user in the database (username = "mno", password = "2222") with the **role ROLE_ADMIN** using the run() method.

---

2. Security Configuration

Inside this method:

```
  @Bean
    public SecurityFilterChain
filterChain(HttpSecurity http) throws Exception {
```

- **Spring Security is told:**
    - Allow anyone to access /authenticate.
    - Require users to have **ADMIN role** to access /home.
    - Disable CSRF (not needed for APIs).
    - Use **JWT** (not sessions) for login.
    - Add a custom filter (JwtRequestFilter) **before** username/password filter.

Also, password encoding and AuthenticationManager are configured.

---

3. User Model and Role

Two model classes:

- **User:** Represents the application user.
- **MyGrantedAuthority:** Represents **roles/authorities** (like ROLE_ADMIN).

Users can have multiple roles (**via @ManyToMany**).

---

4. Login Endpoint (/authenticate)

In AppController.java:

@PostMapping("/authenticate")

This is the **login API**:

- Accepts a JSON body with username and password.
- Tries to authenticate using AuthenticationManager.
- If valid, generates a **JWT token** using the **auth0.jwt library**.
- Adds the token in the **HTTP Header** called Authorization.

Example response header:

Authorization: Bearer <JWT_TOKEN>

---

5. JWT Filter (JwtRequestFilter)

Every request **except /authenticate** goes through this filter:

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter
{
```

- It checks the **Authorization header**.
- If a valid token is found:
  - It extracts the username and roles.
  - Loads user details from the database.
  - Sets user authentication into **Spring Security Context**.

Now the user is considered **logged in** for the request.

---

6. Home API (/home)

```
@GetMapping("/home")
```

- This is a **protected route**.
- Only users with ROLE_ADMIN and a **valid JWT** can access it.
- If the JWT is missing or invalid → Access denied.

---

7. UserDetailsService and UserDetails

- MyUserDetailsService loads user data from the database.
- MyUserDetails converts your User model to something Spring Security understands.

Flow Summary

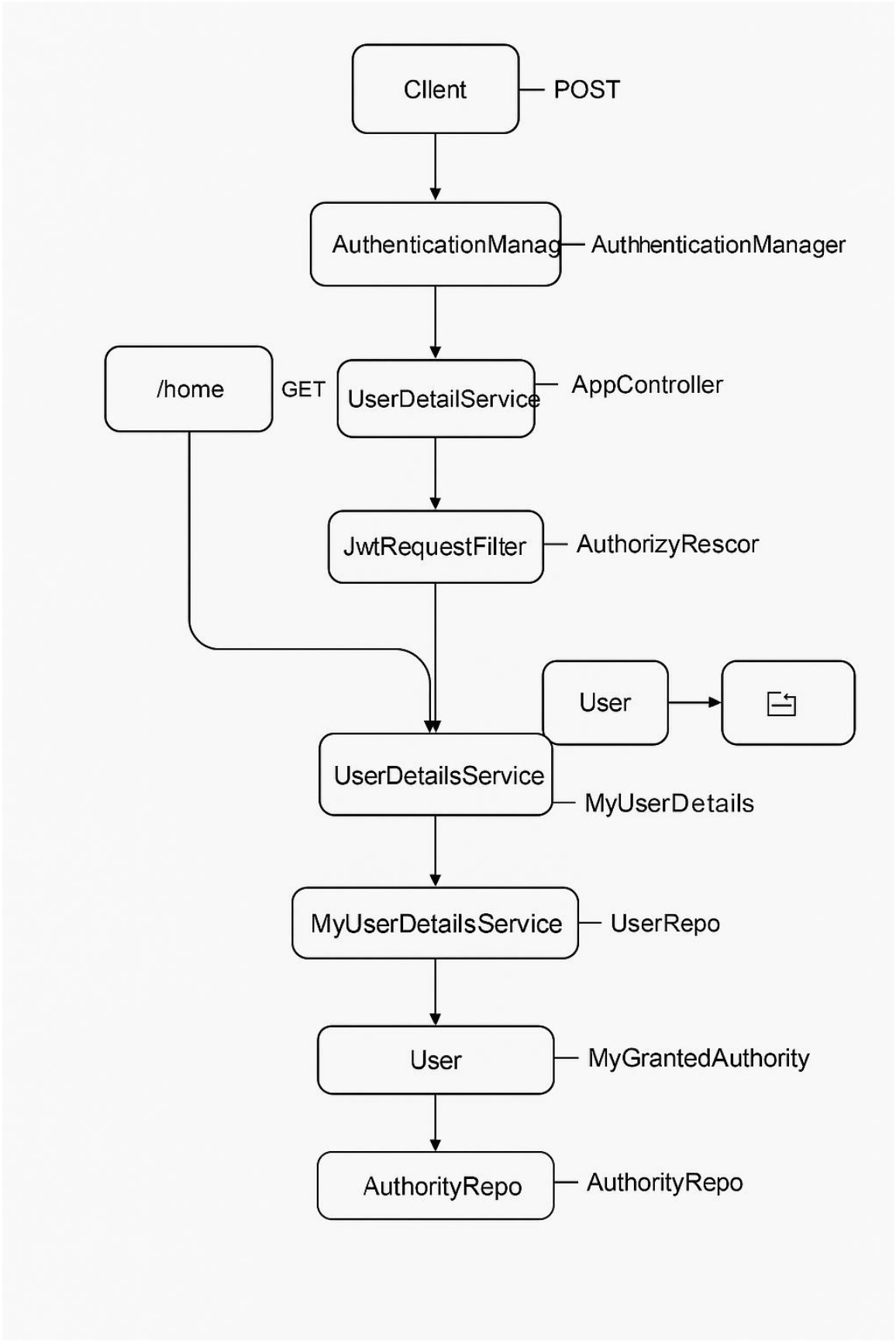
1. User hits POST /authenticate → sends username & password
2. Server authenticates & returns JWT token
3. User sends JWT token in Authorization header for protected routes

4. JwtRequestFilter validates token → sets authentication context
5. If valid, user can access /home (if they have ROLE_ADMIN)

Key Concepts

Concept	Meaning
JWT	A token format that holds user info securely
@Bean	Used to create and manage objects in Spring
AuthenticationManager	Verifies user credentials
FilterChain	Sequence of filters applied to every request
@Autowired	Injects dependencies (classes) automatically
@Service/@Component	Tells Spring to manage these classes
SecurityContextHolder	Holds the security info for the current request

---


CIlent
POST
AuthenticationManag
AuthhenticationManager
/home
GET
UserDetailService
AppController
JwtRequestFilter
AuthorizyRescor
User
UserDetailsService
MyUserDetails
MyUserDetailsService
UserRepo
User
MyGrantedAuthority
AuthorityRepo
AuthorityRepo

Full Flow Breakdown

1. Client (User/Postman) Sends Login Request

- **POST request** sent to /authenticate with username & password.
- Goes to AuthenticationManager for validation.

2. AuthenticationManager

- Uses Spring Security's **AuthenticationManager** to validate user credentials.
- If successful, the controller **(AppController)** creates a JWT token using **com.auth0.jwt.JWT.**

Token sent back to client

- This token must be attached to **future requests** as Authorization: **Bearer <token>.**

Accessing a Protected Endpoint (e.g., /home)

3. Client sends GET request to /home

- With the JWT token in the header.
- This route is **secured**, so token verification is needed.

4. JwtRequestFilter (Custom Filter)

- Intercepts the request before it reaches /home.
- Extracts and verifies the JWT token.
- If valid:
    - Extracts the **username** from the token.

- Passes it to Spring Security for loading user details.

5. UserDetailsService

- Delegates to your custom **service: MyUserDetailsService**.

6. MyUserDetailsService

- Queries the database via **UserRepo** to fetch the User by username.

7. User + Authorities

- The User entity is fetched.
- It contains roles (MyGrantedAuthority) like ROLE_ADMIN.

8. AuthorityRepo

- Used to retrieve or save roles.
- In this flow, it's used during application start-up (CommandLineRunner).

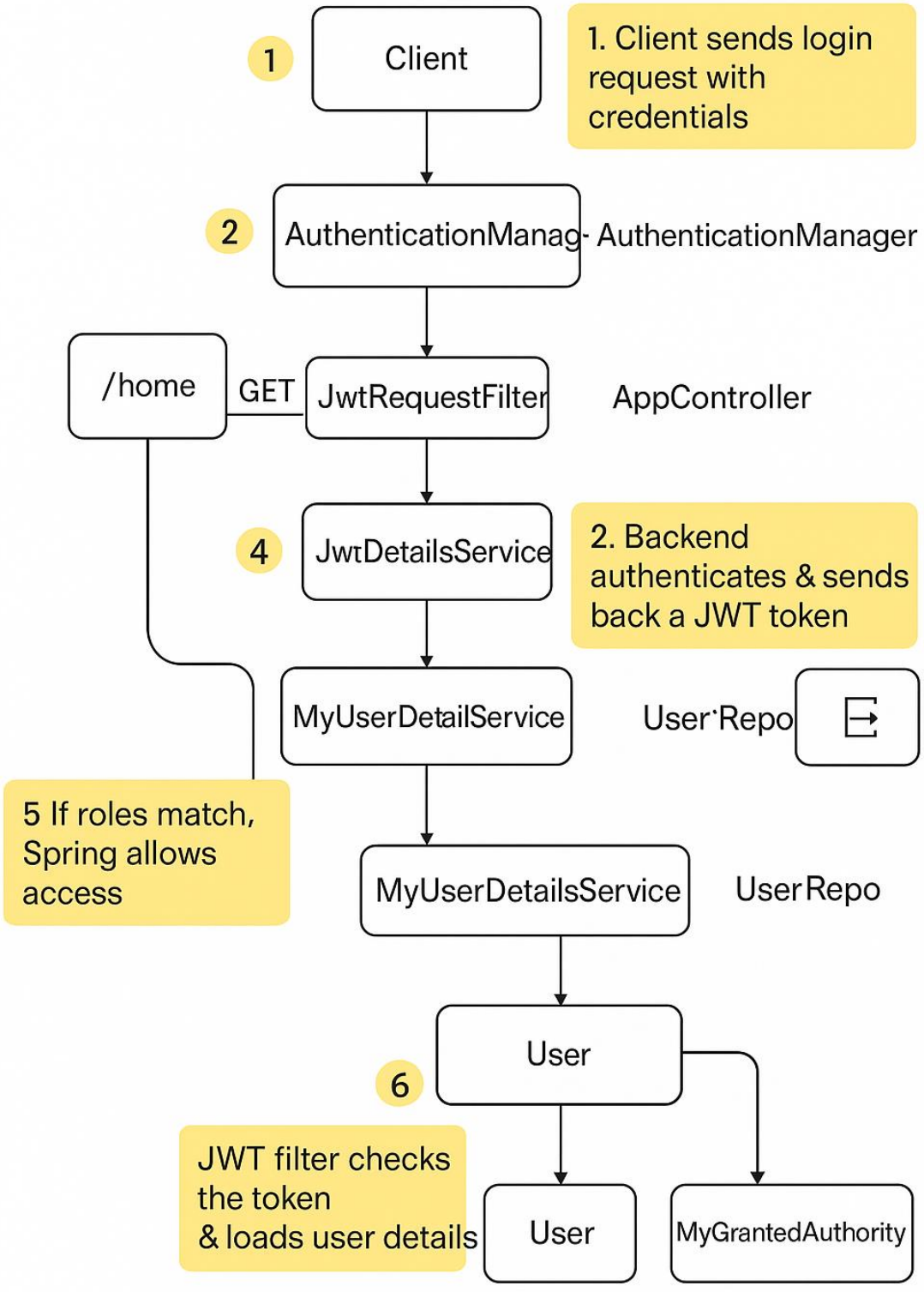
If everything is valid:

- Spring Security sets up an Authentication object.
- Access to /home is **granted** if the user has the required role (ROLE_ADMIN).

Summary in Simple Terms:

Step	Description
1.	Client logs in with credentials.
2.	Backend authenticates and sends back a JWT token.
3.	Client uses this token to access secure endpoints like /home.

4.	JWT filter checks the token and loads user details.
5.	If roles match, Spring allows access. Otherwise, access is denied.


1
Client
1. Client sends login request with credentials
2
AuthenticationManag
AuthenticationManager
/home
GET
JwtRequestFilter
AppController
4
JwtDetailsService
2. Backend authenticates & sends back a JWT token
MyUserDetailService
User Repo
5 If roles match, Spring allows access
MyUserDetailsService
User Repo
User
6
JWT filter checks the token & loads user details
User
MyGrantedAuthority