***Securing REST API in a Healthcare Appointment System***

**Overview**

In modern web applications, securing REST APIs is crucial for protecting resources and data. This tutorial covers the implementation of a secure REST API in a microservice-based architecture using Spring Boot, Spring Security, and JSON Web Tokens (JWT) for authentication and authorization.

**System Architecture**

The system is designed with a clear separation of concerns, as depicted in the provided context and container diagrams. The architecture consists of a front-end client, user and appointment microservices, and their respective databases. Additionally, Kafka is used for asynchronous data transfer, enhancing the system's scalability and decoupling services.

**Context Diagram**

At the highest level, the context diagram illustrates the system's core component, the Appointment System, and its interaction with the User Database, Appointment Database, and Notification System. It also shows the system's primary actors: the Admin, Patient, and Doctor.

**Container Diagram**

The Container Diagram provides a detailed view of the system's architecture, focusing on the interaction between front-end and back-end components. At the core of the front-end is the Micro Front-end Architecture highlighted by the yellow background, which includes:

* **Login Container (ReactJS)**: Serves as the entry point for all user types. Patients, Doctors, and Admins log in through this container.
* **Appointment Container (ReactJS)**: After login, both Patients and Doctors interact with this container to manage and view appointments.
* **Admin Panel Container (ReactJS)**: Post-login, the Admin interacts with this container to access administrative functions.

The **Nginx Load Balancer** sits beneath these containers, efficiently distributing incoming network traffic across the various microservices.

On the back-end side, the diagram illustrates:

* **User Microservice Container**: Handles user-related operations and accesses the **User Database (User DB)**.
* **Appointment Microservice Container**: Manages appointment-related operations, connects to the **Appointment Database (Appointment DB)**, and communicates with the Kafka Container.

The **Kafka Container** is used for handling messaging and data streaming between microservices, ensuring reliable and scalable communication across the system.

**Component Diagram**

The component diagram further breaks down the system into individual components, highlighting the Web Application's structure that consists of:

* A Sign-In Controller for authentication requests.
* Security Components powered by Spring Security for secure access.
* User and Appointment Management components representing the service layer.
* A Notification Component for alerting users about various system events.

**Code Diagram**

The code diagram offers a granular view of the microservice responsible for user authentication and authorization, essential for securing the REST API. This detailed diagram interlinks classes, interfaces, and their respective methods to highlight the operational workflow of user management within the system.

***Users***

This class models the user entity with attributes tailored to store user credentials and roles, such as username, password, and role. It provides the necessary getters and setters for data encapsulation and integrity.

***AuthenticationResponse***

This data transfer object (DTO) encapsulates the JWT and other authentication-related information, serving as the payload in the response after a user successfully authenticates.

***JwtTokenUtil***

Central to JWT handling, this utility class is equipped with methods for token generation (generateToken), validation (validateToken), and extraction of claims (getUsernameFromToken, getRoleFromToken). Its attributes, secretKey and expiration, are pivotal for the creation and verification of secure tokens.

***UsersController***

Acting as the REST API controller, this class defines endpoints for managing user data and handling authentication processes. It leverages services like UsersService and utilities such as JwtTokenUtil to execute operations including user registration (addUser), sign-in (signIn), and role-based data retrieval (getUserByRole).

***UsersService and UsersServiceImpl***

The UsersService interface outlines the contract for user operations, which UsersServiceImpl fulfills, demonstrating a clear separation of concerns in line with service-oriented architecture. The implementation interacts with UserRepository to perform data persistence operations, reflecting a comprehensive approach to user management tasks.

***UserRepository***

This repository interface extends Spring's JpaRepository, suggesting the adoption of Spring Data JPA for database interactions. Its customized methods, such as findByUsername and findByRole, facilitate efficient querying based on user-specific attributes.

***UserMicroserviceApplication***

The starting point of the microservice, housing the main method that launches the Spring Boot application, establishing the foundation for service execution.

***JwtRequestFilter***

This filter class, executing once per API request, is responsible for parsing the Authorization header to validate JWT tokens, ensuring that each request is authenticated before reaching the controller.

***SecurityConfig***

This configuration class sets up the security constraints using Spring Security, defining role-based access controls and the security filter chain that governs API endpoint protection.

***CorsConfig***

This class configures the Cross-Origin Resource Sharing (CORS) policy, which is critical for a microservice architecture where different services may reside on different origins.

Together, these components constitute the backend's security framework, ensuring that user data is managed securely and that only authenticated and authorized users can access the REST API's protected resources.

**Security Implementation**

**JWT Token Utility**

The **JwtTokenUtil** class is at the heart of the JWT handling mechanism. It is responsible for parsing the JWT to extract claims such as the username and roles. It also includes methods to validate the token by checking its expiration.

*public class JwtTokenUtil {*

*private String secretKey;*

*public String getUsernameFromToken(String token) { /\*...\*/ }*

*public String getRoleFromToken(String token) { /\*...\*/ }*

*public Boolean validateToken(String token) { /\*...\*/ }*

*}*

**Security Configuration**

The **SecurityConfig** class configures the HTTP security of the application using Spring Security. It defines which endpoints require authentication and which authorities (roles) are required to access them.

*@Configuration*

*@EnableWebSecurity*

*public class SecurityConfig {*

*public static final String ADMIN = "Admin";*

*public static final String PATIENT = "Patient";*

*public static final String DOCTOR = "Doctor";*

*@Bean*

*public SecurityFilterChain filterChain(HttpSecurity http) throws Exception { /\*...\*/ }*

*}*

**JWT Request Filter**

The **JwtRequestFilter** extends **OncePerRequestFilter** and is used to intercept every request. It retrieves the JWT from the **Authorization** header, validates it, and sets the **SecurityContext** accordingly.

*public class JwtRequestFilter extends OncePerRequestFilter {*

*@Override*

*protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) { /\*...\*/ }*

*}*

**Securing the Endpoints**

The endpoints are secured as per the roles. For example, only Admin can access the user management endpoints, while patients and doctors have restricted access according to their roles.

*http.authorizeHttpRequests((requests) -> requests*

*.requestMatchers(HttpMethod.POST, "/api/users/signin", "/api/users/signup").permitAll()*

*.requestMatchers("/api/users/\*\*").hasAuthority(ADMIN)*

*.requestMatchers("/api/users/find\_role").hasAnyAuthority(ADMIN, PATIENT, DOCTOR)*

*// Other matchers*

*.anyRequest().authenticated());*

**Conclusion**

This tutorial outlined the implementation of a secure REST API within a microservice architecture. We covered the system's structural context, containerization, component breakdown, and detailed the security configuration using Spring Security and JWT. With this setup, the system ensures that API endpoints are protected and only accessible to authenticated and authorized users based on their roles.