Securing a REST API in a Microservices Architecture using JWT and API Gateway

Lup Andrei Florin

Repository

`https://github.com/lupandrei/soa-project`

1 Introduction

In modern distributed systems, security is no longer a secondary concern. When an application is composed of multiple microservices, each exposed through a network, the attack surface grows significantly. A robust and scalable security solution is therefore essential.

This tutorial presents a practical implementation of a secure REST API based on:

- JSON Web Tokens (JWT) for authentication
- Spring Cloud Gateway for centralized authorization
- Nginx for routing and load balancing
- Angular frontend for client-side integration

The goal is to demonstrate how security can be implemented once, at the gateway level, and reused consistently across all microservices.

2 System Architecture

The system follows a typical microservices architecture and contains the following components:

- **User Service** – handles authentication and generates JWT tokens.
- **Restaurant Service** – exposes protected business data.
- **Booking Service** – manages user bookings.

- **Notification Service** – sends asynchronous notifications.
- **API Gateway** – central entry point that enforces security.
- **Nginx** – reverse proxy and load balancer.
- **Angular Frontend** – user interface.

All external requests pass through Nginx and the API Gateway before reaching internal services.

3 Security Design

Instead of securing each microservice individually, authentication and authorization are centralized in the API Gateway. This design provides several advantages:

- consistent security rules across services
- easier maintenance
- reduced code duplication
- improved scalability

The system uses stateless authentication based on JWT tokens. Once authenticated, the client includes the token in every request, allowing the gateway to validate the identity without storing sessions.

4 Authentication Flow

The authentication process follows these steps:

1. The user submits login credentials from the frontend.
2. The User Service validates the credentials.
3. A JWT token is generated and returned to the client.
4. The client stores the token locally.
5. Every subsequent request includes the token in the `Authorization` header.
6. The API Gateway validates the token before forwarding the request.

This approach ensures that only authenticated users can access protected resources.

5 JWT Token Generation

After successful authentication, the User Service creates a JWT token containing the user identity and an expiration time.

```
String token = Jwts.builder()
    .setSubject(user.getEmail())
    .setIssuedAt(new Date())
    .setExpiration(new Date(System.currentTimeMillis() + 3600000))
    .signWith(secretKey, SignatureAlgorithm.HS512)
    .compact();
```

The token is digitally signed, which guarantees its integrity and prevents tampering.

6 Client-Side Integration

On the frontend, Angular stores the token in local storage and automatically attaches it to every outgoing HTTP request using an interceptor.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const token = localStorage.getItem('token');
    if (!token) return next.handle(req);

    const authReq = req.clone({
      setHeaders: { Authorization: `Bearer ${token}` }
    });

    return next.handle(authReq);
  }
}
```

This guarantees that all protected endpoints are accessed securely without requiring manual header management.

7 API Gateway Security Filter

The API Gateway enforces security through a global filter that intercepts every request.

```
@Component
public class JwtAuthGlobalFilter implements GlobalFilter {

    private final JwtUtil jwtUtil;

    @Override
```

```
    public Mono<Void> filter(ServerWebExchange exchange,
        GatewayFilterChain chain) {

        if ("OPTIONS".equalsIgnoreCase(
            exchange.getRequest().getMethodValue())) {
            return chain.filter(exchange);
        }

        String path = exchange.getRequest().getURI().getPath();

        if (path.startsWith("/auth/login") ||
            path.startsWith("/auth/register")) {
            return chain.filter(exchange);
        }

        String authHeader = exchange.getRequest()
            .getHeaders()
            .getFirst(HttpHeaders.AUTHORIZATION);

        if (authHeader == null ||
            !authHeader.startsWith("Bearer ")) {
            exchange.getResponse()
                .setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        String token = authHeader.substring(7);

        try {
            jwtUtil.validateToken(token);
        } catch (Exception e) {
            exchange.getResponse()
                .setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        return chain.filter(exchange);
    }
}
```

Only authenticated requests are allowed to reach internal services.

8 JWT Validation

The validation logic ensures that the token was signed with the correct secret key.

```
@Component
```

```
public class JwtUtil {

    @Value("${jwt.secret}")
    private String SECRET;

    private SecretKey KEY;

    @PostConstruct
    public void init() {
        this.KEY = Keys.hmacShaKeyFor(
            SECRET.getBytes(StandardCharsets.UTF_8));
    }

    public Claims validateToken(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(KEY)
            .build()
            .parseClaimsJws(token)
            .getBody();
    }
}
```

A shared secret is injected through Docker environment variables, ensuring consistency between services.

9 Testing and Validation

The security implementation was validated through both client-side and server-side testing in order to ensure that unauthorized access is consistently prevented.

Frontend validation

At the client level, the Angular application enforces access control using route guards. When a user attempts to access the `/restaurants` page without being authenticated, the system automatically redirects the user to the login page. This ensures that protected pages cannot be accessed directly through the browser without a valid session.

This behavior confirms that security is enforced not only at the API level, but also at the user interface level, improving the overall user experience and reducing accidental unauthorized access.

Backend validation

At the server level, security was tested using direct HTTP requests through Postman. When attempting to access protected endpoints such as:

- `GET /restaurants`
- `POST /bookings`

without providing a valid `Authorization` header, the API Gateway consistently responds with:

```
401 Unauthorized
```

This confirms that the gateway correctly intercepts and blocks unauthorized requests before they reach internal microservices.

When a valid JWT token is included in the request header:

```
Authorization:  Bearer <token>
```

the request is successfully forwarded to the target service, proving that authentication and authorization are functioning as intended.

End-to-end validation

Finally, end-to-end testing was performed by logging in through the frontend, obtaining a JWT token, and accessing protected resources such as the restaurant list. The successful flow demonstrates that security is consistently enforced across all layers of the system: frontend, gateway, and backend services.

10 Benefits of the Approach

The implemented solution provides:

- centralized authentication and authorization
- stateless and scalable security
- simplified microservice design
- easy integration with frontend applications

11 Conclusion

This tutorial demonstrated how a REST API can be effectively secured in a microservices environment using JWT and an API Gateway. By centralizing security at the gateway level and combining it with a modern frontend and container-based deployment, the system achieves both robustness and scalability. The presented approach reflects industry best practices and can serve as a strong foundation for real-world applications.