**How to Secure Your Microservices with JWT Authentication Using Spring Cloud Gateway**

In this tutorial, we will explore how to secure microservices using **JWT Authentication** with **Spring Cloud Gateway**. The architecture discussed will include one microservice acting as an **API Gateway** that handles **centralized authentication** and routes incoming requests to downstream services. This approach provides an easy way to scale by adding multiple microservices while managing authentication and authorization centrally.

**Key Features of the Architecture:**

1. **API Gateway as Central Auth Service**: All requests will pass through the Gateway, which will authenticate the user and route traffic to microservices based on roles and privileges.
2. **JWT-Based Security**: The API Gateway will generate JWT tokens upon successful login using the /authenticate endpoint. The token is then validated for all subsequent requests.
3. **Gateway Filter for Authentication**: We will implement a **Gateway Filter** that ensures every request is authenticated. It validates the JWT token and extracts user roles, passing them along as headers to downstream services.
4. **Role-Based Authorization**: Downstream microservices will perform role-based authorization using the role information provided by the Gateway, ensuring that only authorized users can access certain endpoints.

**How It Works:**

1. **User Authentication**: A user provides their username and password via the /authenticate endpoint. The API Gateway generates a JWT token containing the user's details and roles.
2. **Token Validation**: For all subsequent requests, the Gateway's filter validates the JWT token to ensure it’s valid and unexpired. It extracts user roles and attaches them as headers to each request.
3. **Routing & Role Propagation**: The Gateway routes the request to the appropriate microservice, passing along user role information. Downstream services then use the role details for **@PreAuthorize** checks and role-based access control.

**Benefits of this Architecture:**

* **Centralized Security**: Authentication and authorization are handled at the Gateway level, reducing complexity for individual microservices.
* **Scalable**: Easily add new microservices without the need for additional security configurations.
* **Role-Based Control**: Microservices receive user role information from the Gateway, enabling fine-grained access control for different endpoints.

**Prompt: Develop JWT Authentication with Spring Cloud Gateway for Securing Microservices**

Your task is to build a secure microservices architecture using **Spring Cloud Gateway** with **JWT Authentication**. The system should have a central API Gateway for user authentication and routing requests to downstream services, ensuring that each service is protected via role-based access control.

**Requirements:**

1. **User Authentication**:
   * Implement a /authenticate endpoint in the API Gateway where users will provide their username and password.
   * Generate a **JWT token** upon successful authentication, including the user's details (username, roles).
   * Return the JWT token to the user for subsequent requests.
2. **Gateway Filter for Token Validation**:
   * Implement a **Gateway Filter** that intercepts all requests, except for /authenticate.
   * The filter should validate the **JWT token** from the request header, ensuring it is valid and unexpired.
   * Extract the user's roles from the token and forward this information in the headers to downstream services.
3. **Routing**:
   * Configure the Gateway to route authenticated requests to the appropriate downstream microservices.
   * Include user role details in the headers when routing to downstream services.
4. **Downstream Services**:
   * Each downstream microservice should implement **role-based authorization** using Spring’s @PreAuthorize annotation.
   * Ensure that services can access the user's role details from the headers to enforce access control (e.g., Admin, User roles).
5. **Role-Based Access**:
   * Set up a system where specific routes in downstream services are protected based on user roles (e.g., Admins can perform all actions, Users can only read).
6. **Stateless Security**:
   * The entire architecture should be stateless, meaning no session management is required. All security-related information will be contained within the JWT.

**Technologies:**

* **Spring Cloud Gateway** for API routing and filtering.
* **JWT** for secure token-based authentication.
* **Spring Security** for role-based authorization in microservices.
* **Spring WebFlux** for the reactive setup in the API Gateway.
* **Spring Boot** for the downstream microservices.

**Steps:**

1. Set up a **Spring Boot** project for the **API Gateway**.
2. Add a **JWT Authentication Filter** in the Gateway to validate tokens and route requests.
3. Set up a /authenticate endpoint that generates JWT tokens.
4. Create downstream **Spring Boot** services that handle **role-based authorization** using @PreAuthorize.
5. Ensure the Gateway passes JWT and role information as headers to these downstream services.
6. Test the architecture by making authenticated requests and checking if role-based access works as expected.

By completing this prompt, you’ll have a fully functional microservices architecture with **centralized authentication** and **role-based authorization** using **JWT tokens** and **Spring Cloud Gateway**.

User Prompt :As a java senior developer Develop JWT Authentication with Spring Cloud Gateway for Securing Microservices Your task is to build a secure microservices architecture using Spring Cloud Gateway with JWT Authentication. The system should have a central API Gateway for user authentication and routing requests to downstream services, ensuring that each service is protected via role-based access control. User auth microservice will have enpoints to authenticate and validate jwt.Develop a Gateway filter and apply this for each route.

**1. Create the User Auth Microservice**

This microservice will handle user authentication and JWT token validation. It will have two key endpoints:

* /authenticate: For user login, generating the JWT.
* /validate: For JWT validation, used by the API Gateway.

**Dependencies:**

Add these dependencies in your pom.xml:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.1</version>

</dependency>

**JWT Utility Class:**

java

Copy code

import io.jsonwebtoken.Claims;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.stereotype.Component;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Component

public class JwtUtil {

private String secret = "mySecretKey";

public String generateToken(String username, String role) {

Map<String, Object> claims = new HashMap<>();

claims.put("role", role);

return createToken(claims, username);

}

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder()

.setClaims(claims)

.setSubject(subject)

.setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour

.signWith(SignatureAlgorithm.HS256, secret)

.compact();

}

public Claims extractClaims(String token) {

return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();

}

public boolean validateToken(String token, String username) {

final String extractedUsername = extractUsername(token);

return (extractedUsername.equals(username) && !isTokenExpired(token));

}

public String extractUsername(String token) {

return extractClaims(token).getSubject();

}

private Boolean isTokenExpired(String token) {

return extractClaims(token).getExpiration().before(new Date());

}

}

**AuthController:**

java

Copy code

@RestController

@RequestMapping("/auth")

public class AuthController {

@Autowired

private JwtUtil jwtUtil;

@PostMapping("/authenticate")

public ResponseEntity<?> createToken(@RequestBody AuthRequest authRequest) {

// Validate the user (this is an example, replace with real user validation)

if ("user".equals(authRequest.getUsername()) && "password".equals(authRequest.getPassword())) {

String token = jwtUtil.generateToken(authRequest.getUsername(), "ROLE\_USER");

return ResponseEntity.ok(new AuthResponse(token));

} else {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();

}

}

@PostMapping("/validate")

public ResponseEntity<?> validateToken(@RequestHeader("Authorization") String token) {

// Validate the JWT token

try {

Claims claims = jwtUtil.extractClaims(token.replace("Bearer ", ""));

return ResponseEntity.ok(claims);

} catch (Exception e) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid Token");

}

}

}

class AuthRequest {

private String username;

private String password;

// getters and setters

}

class AuthResponse {

private String token;

// constructor, getters, and setters

}

**2. Set Up Spring Cloud Gateway**

Your **API Gateway** will act as the central service to authenticate and route requests to downstream microservices. It will call the /validate endpoint to check JWT tokens and route the request based on roles.

**Add Dependencies:**

xml

Copy code

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-webflux</artifactId>

</dependency>

**Gateway Filter for Token Validation:**

java

Copy code

@Component

public class JwtAuthenticationFilter implements GlobalFilter {

@Autowired

private WebClient.Builder webClientBuilder;

@Override

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

ServerHttpRequest request = exchange.getRequest();

if (request.getURI().getPath().equals("/auth/authenticate")) {

return chain.filter(exchange); // Allow open access to the authentication endpoint

}

String authorizationHeader = request.getHeaders().getFirst(HttpHeaders.AUTHORIZATION);

if (authorizationHeader == null || !authorizationHeader.startsWith("Bearer ")) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);

return exchange.getResponse().setComplete();

}

String token = authorizationHeader.replace("Bearer ", "");

return webClientBuilder.build()

.post()

.uri("http://auth-service/auth/validate")

.header(HttpHeaders.AUTHORIZATION, authorizationHeader)

.retrieve()

.bodyToMono(Map.class)

.flatMap(response -> {

String role = (String) response.get("role");

// Add roles and other user info to headers before forwarding

ServerHttpRequest modifiedRequest = exchange.getRequest().mutate()

.header("X-Role", role)

.build();

return chain.filter(exchange.mutate().request(modifiedRequest).build());

})

.onErrorResume(e -> {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);

return exchange.getResponse().setComplete();

});

}

}

**Gateway Configuration:**

yaml

Copy code

spring:

cloud:

gateway:

routes:

- id: user-service

uri: http://user-service

predicates:

- Path=/user/\*\*

filters:

- StripPrefix=1

**3. Downstream Microservices**

Each microservice will receive the role information in the request headers and enforce role-based authorization. Example:

java

Copy code

@RestController

public class UserController {

@GetMapping("/user/data")

@PreAuthorize("hasRole('ROLE\_USER')")

public String getUserData() {

return "User data";

}

}

**Enable Security in Downstream Services:**

java

Copy code

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.anyRequest().authenticated();

}

}

**Conclusion**

With this setup:

* The **API Gateway** manages authentication by calling the **User Auth Service**.
* JWT tokens are validated using a custom **Gateway Filter**.
* Downstream services implement **role-based authorization** using headers passed from the Gateway.

This architecture ensures that all microservices are secured and roles are enforced centrally.

4o

**You said:**

write unit test cases appropriately

**ChatGPT said:**

ChatGPT

Here’s how to write **unit test cases** for the key components of the JWT authentication and routing mechanism in your Spring Cloud Gateway microservices architecture.

**1. Unit Test for JwtUtil**

This is the utility class responsible for generating and validating JWT tokens. We’ll mock the JwtUtil class behavior to test its key methods.

**Test Dependencies:**

In your pom.xml, add the dependencies for JUnit, Mockito, and Spring Test:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<scope>test</scope>

</dependency>

**JwtUtilTest:**

java

Copy code

import io.jsonwebtoken.Claims;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.MockitoAnnotations;

import static org.junit.jupiter.api.Assertions.\*;

class JwtUtilTest {

@InjectMocks

private JwtUtil jwtUtil;

@BeforeEach

public void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void generateToken\_ShouldReturnValidToken() {

String token = jwtUtil.generateToken("user", "ROLE\_USER");

assertNotNull(token);

}

@Test

void validateToken\_ShouldReturnTrue\_ForValidToken() {

String token = jwtUtil.generateToken("user", "ROLE\_USER");

assertTrue(jwtUtil.validateToken(token, "user"));

}

@Test

void extractUsername\_ShouldReturnCorrectUsername() {

String token = jwtUtil.generateToken("user", "ROLE\_USER");

assertEquals("user", jwtUtil.extractUsername(token));

}

@Test

void extractClaims\_ShouldReturnClaims() {

String token = jwtUtil.generateToken("user", "ROLE\_USER");

Claims claims = jwtUtil.extractClaims(token);

assertEquals("ROLE\_USER", claims.get("role"));

}

@Test

void isTokenExpired\_ShouldReturnFalse\_ForValidToken() {

String token = jwtUtil.generateToken("user", "ROLE\_USER");

assertFalse(jwtUtil.extractClaims(token).getExpiration().before(new Date()));

}

}

**2. Unit Test for AuthController**

Test the endpoints of the **AuthController** to ensure proper authentication and token validation. We will mock the JwtUtil to simulate the behavior without actually generating or validating tokens.

**AuthControllerTest:**

java

Copy code

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import static org.junit.jupiter.api.Assertions.\*;

import static org.mockito.Mockito.\*;

class AuthControllerTest {

@InjectMocks

private AuthController authController;

@Mock

private JwtUtil jwtUtil;

@BeforeEach

public void setUp() {

MockitoAnnotations.openMocks(this);

}

@Test

void authenticate\_ShouldReturnToken\_ForValidCredentials() {

AuthRequest request = new AuthRequest();

request.setUsername("user");

request.setPassword("password");

when(jwtUtil.generateToken(anyString(), anyString())).thenReturn("mockedToken");

ResponseEntity<?> response = authController.createToken(request);

assertEquals(HttpStatus.OK, response.getStatusCode());

AuthResponse authResponse = (AuthResponse) response.getBody();

assertNotNull(authResponse);

assertEquals("mockedToken", authResponse.getToken());

}

@Test

void authenticate\_ShouldReturnUnauthorized\_ForInvalidCredentials() {

AuthRequest request = new AuthRequest();

request.setUsername("user");

request.setPassword("wrongpassword");

ResponseEntity<?> response = authController.createToken(request);

assertEquals(HttpStatus.UNAUTHORIZED, response.getStatusCode());

}

@Test

void validateToken\_ShouldReturnValidClaims\_ForValidToken() {

String token = "mockedToken";

Claims claims = mock(Claims.class);

when(jwtUtil.extractClaims(anyString())).thenReturn(claims);

ResponseEntity<?> response = authController.validateToken("Bearer " + token);

assertEquals(HttpStatus.OK, response.getStatusCode());

assertEquals(claims, response.getBody());

}

@Test

void validateToken\_ShouldReturnUnauthorized\_ForInvalidToken() {

when(jwtUtil.extractClaims(anyString())).thenThrow(new RuntimeException("Invalid token"));

ResponseEntity<?> response = authController.validateToken("Bearer invalidToken");

assertEquals(HttpStatus.UNAUTHORIZED, response.getStatusCode());

assertEquals("Invalid Token", response.getBody());

}

}

**3. Unit Test for JwtAuthenticationFilter**

Since the **JwtAuthenticationFilter** is a component in the Spring Cloud Gateway, we can use **WebFluxTest** for testing its behavior. Here we will mock the WebClient and validate the filter’s behavior for authorized and unauthorized requests.

**JwtAuthenticationFilterTest:**

java

Copy code

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.server.reactive.ServerHttpRequest;

import org.springframework.web.reactive.function.client.WebClient;

import org.springframework.web.server.ServerWebExchange;

import reactor.core.publisher.Mono;

import static org.junit.jupiter.api.Assertions.\*;

import static org.mockito.Mockito.\*;

class JwtAuthenticationFilterTest {

@InjectMocks

private JwtAuthenticationFilter jwtAuthenticationFilter;

@Mock

private WebClient.Builder webClientBuilder;

@Mock

private WebClient webClient;

@Mock

private WebClient.RequestHeadersSpec<?> requestHeadersSpec;

@Mock

private WebClient.ResponseSpec responseSpec;

@BeforeEach

public void setUp() {

MockitoAnnotations.openMocks(this);

when(webClientBuilder.build()).thenReturn(webClient);

when(webClient.post()).thenReturn(requestHeadersSpec);

}

@Test

void filter\_ShouldAllowUnauthenticatedAccess\_ForAuthenticateEndpoint() {

ServerWebExchange exchange = mock(ServerWebExchange.class);

ServerHttpRequest request = mock(ServerHttpRequest.class);

when(exchange.getRequest()).thenReturn(request);

when(request.getURI()).thenReturn(URI.create("/auth/authenticate"));

Mono<Void> result = jwtAuthenticationFilter.filter(exchange, null);

assertNotNull(result);

}

@Test

void filter\_ShouldRejectUnauthorizedRequest\_WhenTokenIsInvalid() {

ServerWebExchange exchange = mock(ServerWebExchange.class);

ServerHttpRequest request = mock(ServerHttpRequest.class);

when(exchange.getRequest()).thenReturn(request);

when(request.getHeaders()).thenReturn(new HttpHeaders());

jwtAuthenticationFilter.filter(exchange, null)

.subscribe(null, ex -> assertTrue(ex instanceof IllegalStateException));

}

@Test

void filter\_ShouldValidateTokenAndForwardRequest\_WhenTokenIsValid() {

ServerWebExchange exchange = mock(ServerWebExchange.class);

ServerHttpRequest request = mock(ServerHttpRequest.class);

HttpHeaders headers = new HttpHeaders();

headers.add(HttpHeaders.AUTHORIZATION, "Bearer validToken");

when(exchange.getRequest()).thenReturn(request);

when(request.getHeaders()).thenReturn(headers);

when(requestHeadersSpec.header(HttpHeaders.AUTHORIZATION, "Bearer validToken"))

.thenReturn(requestHeadersSpec);

when(requestHeadersSpec.retrieve()).thenReturn(responseSpec);

when(responseSpec.bodyToMono(Map.class)).thenReturn(Mono.just(Map.of("role", "ROLE\_USER")));

Mono<Void> result = jwtAuthenticationFilter.filter(exchange, null);

assertNotNull(result);

}

}

**4. Integration Testing with Spring Boot Test**

Once your individual unit tests are in place, you can write integration tests for the entire flow of the API Gateway interacting with downstream services.

**Example:**

java

Copy code

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class GatewayIntegrationTest {

@Autowired

private TestRestTemplate restTemplate;

@Test

public void whenAuthenticate\_thenReturnToken() {

// Call /auth/authenticate and validate JWT token

ResponseEntity<AuthResponse> response = restTemplate.postForEntity(

"/auth/authenticate", new AuthRequest("user", "password"), AuthResponse.class);

assertEquals(HttpStatus.OK, response.getStatusCode());

assertNotNull(response.getBody().getToken());

}

@Test

public void whenAccessUserServiceWithValidToken\_thenAllowAccess() {

String token = authenticateAndGetToken();

HttpHeaders headers = new HttpHeaders();

headers.set("Authorization", "Bearer " + token);

HttpEntity<Void> request = new HttpEntity<>(headers);

ResponseEntity<String> response = restTemplate.exchange(

"/user/data", HttpMethod.GET, request, String.class);

assertEquals(HttpStatus.OK, response.getStatusCode());

}

private String authenticateAndGetToken() {

ResponseEntity<AuthResponse> response = restTemplate.postForEntity(

"/auth/authenticate", new AuthRequest("user", "password"), AuthResponse.class);

return response.getBody().getToken();

}

}

**Conclusion:**

These unit tests cover the essential pieces of your JWT-based authentication system. The JwtUtil class, the authentication controller, and the filter are tested with mocks. Integration tests ensure that the components work together seamlessly.

Implementing secure headers and ensuring HTTPS are essential for securing communication between your \*\*Node.js\*\* and \*\*Spring Boot\*\* applications. Here’s how you can achieve both:

---

### 1. \*\*Use Secure Headers in Node.js\*\*

To ensure that headers are secure and less likely to be tampered with, follow these steps:

#### a) Use Custom, Non-Standard Header Names

- When passing user claims (like roles) from Node.js to Spring Boot, avoid using common or standard HTTP headers (`Authorization`, `Cookie`, etc.).

- Create custom header names for sensitive claims that are less predictable and harder for external clients to guess or spoof.

- Example of custom headers:

```javascript

const options = {

headers: {

'X-App-User-ID': userId, // User ID

'X-App-Roles': roles.join(','), // Comma-separated roles

'X-App-Client-ID': clientId // Optional client-specific identifier

}

};

```

#### b) Add Integrity or Source Verification to Headers

- Use a \*\*HMAC (Hash-based Message Authentication Code)\*\* to sign headers for integrity verification:

- Generate an HMAC using a shared secret key and include it as a header (`X-Signature`, for example).

- In Spring Boot, verify the HMAC against the received headers to confirm authenticity.

- Example in Node.js:

```javascript

const crypto = require('crypto');

const secretKey = 'shared\_secret\_key';

const signature = crypto

.createHmac('sha256', secretKey)

.update(userId + roles.join(','))

.digest('hex');

const options = {

headers: {

'X-App-User-ID': userId,

'X-App-Roles': roles.join(','),

'X-Signature': signature

}

};

```

- \*\*In Spring Boot\*\*, use the same secret key to verify the HMAC and confirm data integrity.

#### c) Apply Header Validation Rules in Spring Boot

- In Spring Boot, validate headers by creating a filter or middleware that:

- \*\*Checks for required headers\*\*: Ensure headers like `X-App-User-ID` and `X-App-Roles` are present in each request.

- \*\*Verifies header format\*\*: Enforce strict validation, such as regex or format rules, to make it harder for malicious headers to slip through.

- \*\*Validates the HMAC signature\*\*: Confirm that the signature is correct, ensuring data was sent by your Node.js service and not modified.

---

### 2. \*\*Use HTTPS for Secure Communication\*\*

To protect data in transit, ensure that \*\*both Node.js and Spring Boot communicate over HTTPS\*\*:

#### a) Set Up HTTPS in Node.js

- If Node.js is the entry point to your application, configure it to accept only HTTPS connections.

- Use TLS certificates (e.g., from Let’s Encrypt for production environments or self-signed certificates for development).

- Example HTTPS setup in Node.js:

```javascript

const https = require('https');

const fs = require('fs');

const express = require('express');

const app = express();

// Load SSL certificate and key

const options = {

key: fs.readFileSync('/path/to/your/private-key.pem'),

cert: fs.readFileSync('/path/to/your/certificate.pem')

};

// Create HTTPS server

https.createServer(options, app).listen(443, () => {

console.log('Server is running on HTTPS');

});

```

#### b) Enforce HTTPS in Spring Boot

- In Spring Boot, configure HTTPS in `application.properties` or `application.yml` by specifying an SSL certificate.

- Example configuration in `application.properties`:

```properties

server.port=8443

server.ssl.enabled=true

server.ssl.key-store=classpath:keystore.jks

server.ssl.key-store-password=yourpassword

server.ssl.key-password=yourpassword

```

- \*\*Generate a Keystore\*\*: If you don't have a keystore, create one using `keytool`:

```bash

keytool -genkey -alias springboot -keyalg RSA -keysize 2048 -keystore keystore.jks -validity 365

```

#### c) Enforce HTTPS Between Node.js and Spring Boot

- Ensure that Node.js communicates with Spring Boot only over HTTPS by setting Spring Boot's URL to start with `https://`.

- If both are hosted within a secure network or Kubernetes cluster, set up a mutual TLS (mTLS) connection to further enhance security.

---

### 3. \*\*Additional Security Configurations\*\*

#### a) Enable HTTP Strict Transport Security (HSTS)

- In Node.js, use the \*\*`helmet`\*\* library to enable HSTS, which enforces HTTPS for future requests.

- Example:

```javascript

const helmet = require('helmet');

app.use(helmet.hsts({ maxAge: 31536000, includeSubDomains: true }));

```

#### b) Restrict CORS

- In Node.js, configure CORS to allow only specific origins and headers.

- Example:

```javascript

const cors = require('cors');

app.use(cors({

origin: 'https://your-trusted-origin.com',

allowedHeaders: ['X-App-User-ID', 'X-App-Roles', 'X-Signature'],

methods: ['GET', 'POST']

}));

```

#### c) API Gateway or Load Balancer (Optional)

- If you’re using an API Gateway or load balancer (like NGINX or Azure API Management), set up HTTPS termination at the gateway level and enforce secure headers.

---

### Summary

By combining secure headers, HMAC validation, and HTTPS, you establish a secure communication layer between Node.js and Spring Boot:

- Use custom header names and sign headers for integrity.

- Enforce HTTPS in both Node.js and Spring Boot.

- Use HSTS, CORS restrictions, and optionally an API Gateway for added security.

These practices help ensure that only trusted, verified requests reach your Spring Boot application.Top of Form

Bottom of Form