Implementação de API HMAC com JWT e Segurança em Spring

Este documento fornece uma explicação técnica detalhada sobre a implementação de uma API em Spring que utiliza JSON Web Tokens (JWT) e HMAC para autenticação e validação de requisições.

A implementação é composta por três classes principais:

- AuthController,
- HMACController
- SecurityConfig.

A seguir, analisaremos cada uma delas, detalhando suas funcionalidades e a lógica de cada método.

Estrutura do Projeto

O projeto é estruturado nas seguintes classes:

a) ***AuthController****: gerencia a autenticação de usuários e a geração de tokens JWT.*
b) ***HMACController****: gerencia a validação de HMAC e a geração de tokens JWT.*
c) ***SecurityConfig****: contém as configurações de segurança da API.*

Dependências Necessárias

Para utilizar JWT e HMAC, é imprescindível incluir as seguintes dependências em seu pom.xml:

xml
```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

1. Classe AuthController

A classe AuthController é responsável pela autenticação de usuários e geração de tokens JWT.

1.1. Anotações

- *@**RestController**: Indica que esta classe é um controlador de API REST, o que permite que os métodos da classe sejam expostos como endpoints HTTP.*
- *@**RequestMapping("/api/auth")**: Define a URL base para as rotas deste controlador. Todos os endpoints dentro deste controlador terão a URL iniciando com /api/auth.*

1.2. Campos

- ***SECRET_KEY**: Chave secreta utilizada para assinar os tokens. **Importante**: a chave deve ser complexa e mantida em segurança, preferencialmente em um sistema de gerenciamento de segredos.*
- ***EXPIRATION_TIME**: Define o tempo de expiração do token em milissegundos (10 dias), o que pode ser ajustado de acordo com a política de segurança da sua aplicação.*

1.3. Método login

Este método é um endpoint que aceita requisições POST e recebe as credenciais do usuário:

```
@PostMapping("/login")
public String login(@RequestBody UserCredentials credentials) {
    if ("usuario".equals(credentials.getUsername()) &&
"senha".equals(credentials.getPassword())) {
        return generateToken(credentials.getUsername());
    } else {
        throw new RuntimeException("Credenciais inválidas");
    }
}
```

Detalhes do Método

- ***Parâmetro @RequestBody UserCredentials credentials**: O objeto UserCredentials contém o nome de usuário e a senha enviados pelo cliente no corpo da requisição.*
- ***Validação de Credenciais**: O método verifica se as credenciais correspondem a valores hardcoded. Para um sistema real, esta lógica deve ser substituída por uma validação contra um banco de dados.*
- ***Geração de Token**: Se as credenciais forem válidas, o método chama generateToken(username) e retorna o token JWT gerado.*

1.4. Método generateToken

Gera um token JWT assinado com a chave secreta:

java
```
private String generateToken(String username) {
    return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
            .compact();
}
```

Detalhes do Método

- **setSubject(username)**: Define o nome de usuário como o sujeito do token.
- **setIssuedAt(new Date())**: Define a data e hora em que o token foi emitido.
- **setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))**: Define a data de expiração do token, que é calculada somando o tempo de expiração ao momento atual.
- **signWith(SignatureAlgorithm.HS256, SECRET_KEY)**: Assina o token usando o algoritmo HS256 com a chave secreta fornecida.
- **compact()**: Compacta o token em uma string que pode ser retornada ao cliente

1.5. Classe Interna UserCredentials

Esta classe encapsula as credenciais do usuário:

java
```
static class UserCredentials {
    private String username;
    private String password;

    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }
    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

Detalhes da Classe

- **Encapsulamento**: A classe possui métodos getter e setter para acessar e modificar as propriedades username e password.

2. Classe HMACController

A classe HMACController gerencia a validação de HMAC e a geração de tokens.

2.1. Anotações

- **@RestController**: Como na classe anterior, define que esta classe é um controlador REST.
- **@RequestMapping("/api")**: Define a URL base para os endpoints deste controlador, que começará com /api.

2.2. Campos

- **SECRET_KEY**: Chave secreta utilizada para calcular o HMAC.
- **MOCK_USERNAME e MOCK_PASSWORD**: Credenciais mockadas para testes, que devem ser substituídas por uma lógica de autenticação real em produção.

2.3. Método authenticateUser

Este método é um endpoint que aceita requisições POST e gera um token JWT após validar as credenciais:

```
@PostMapping("/token/login")
public String authenticateUser(@RequestBody LoginRequest loginRequest) {
    if (isValidUser(loginRequest.getUsername(), loginRequest.getPassword())) {
        String token = generateToken(loginRequest.getUsername());
        logger.info("User '{}' authenticated. Token generated: {}", loginRequest.getUsername(),
token);
        return "Login successful! Token: " + token;
    } else {
        return "Invalid username or password!";
    }
}
```

Detalhes do Método

- **Parâmetro @RequestBody LoginRequest loginRequest**: O objeto LoginRequest contém o nome de usuário e a senha enviados pelo cliente.
- **Validação de Credenciais**: O método isValidUser é chamado para validar as credenciais.
- **Geração de Token**: Se as credenciais forem válidas, um token JWT é gerado e retornado ao cliente.
- **Log de Autenticação**: A operação de autenticação é registrada utilizando o Logger.

2.4. Método isValidUser

Este método valida as credenciais do usuário com base nos valores mockados:

```
private boolean isValidUser(String username, String password) {
    return MOCK_USERNAME.equals(username) && MOCK_PASSWORD.equals(password);
}
```

Detalhes do Método

- **Comparação de Credenciais**: Verifica se o nome de usuário e a senha correspondem aos valores mockados. Para integração com um banco de dados, você deve implementar a lógica apropriada.

2.5. Método generateToken

Similar ao método da classe AuthController, este método gera um token JWT:

java

```
private String generateToken(String username) {
    return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 120000)) // Token válido por 120
segundos
            .signWith(SignatureAlgorithm.HS256, SECRET_KEY.getBytes())
            .compact();
}
```

Detalhes do Método

- **Tempo de Expiração**: Para facilitar testes, o token gerado é válido por 120 segundos.
- **Assinatura**: Utiliza a mesma lógica de assinatura que na classe anterior.

2.6. Método validateHMAC

Este método valida o HMAC fornecido pelo cliente:

```
@PostMapping("/secure-data")
public String validateHMAC(@RequestBody SecureRequest requestData,
@RequestHeader("HMAC") String clientHMAC) {
    try {
        String serverHMAC = calculateHMAC(requestData.getMessage(), SECRET_KEY);

        if (isRequestExpired(requestData.getTimestamp())) {
            return "Request expired.";
```

```
        }

        if (serverHMAC.equals(clientHMAC)) {
            return "HMAC validation successful!";
        } else {
            return "Invalid HMAC!";
        }
    } catch (Exception e) {
        return "Error during HMAC validation!";
    }
}
```

Detalhes do Método

- **Parâmetros**:
  - @RequestBody SecureRequest requestData: Contém os dados a serem validados e o timestamp da requisição.
  - @RequestHeader("HMAC") String clientHMAC: O HMAC enviado pelo cliente para validação.
- **Cálculo do HMAC**: O HMAC é calculado utilizando o método calculateHMAC.
- **Validação do Timestamp**: O método isRequestExpired verifica se a requisição está expirada.
- **Comparação do HMAC**: Se o HMAC calculado no servidor corresponder ao HMAC do cliente, a validação é bem-sucedida.

2.7. Método calculateHMAC

Calcula o HMAC da mensagem recebida:

java

```
private String calculateHMAC(String data, String key) throws Exception {
    SecretKeySpec secretKey = new SecretKeySpec(key.getBytes(), "HmacSHA256");
    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(secretKey);
    byte[] hmacData = mac.doFinal(data.getBytes());
    return Base64.getEncoder().encodeToString(hmacData);
}
```

Detalhes do Método

- **Criação da Chave Secreta**: Utiliza SecretKeySpec para criar uma chave secreta baseada na string key.
- **Instância do Mac**: Inicializa um objeto Mac com o algoritmo HmacSHA256.
- **Cálculo do HMAC**: O HMAC é calculado a partir da string de entrada e convertido em Base64 para facilitar o transporte.

2.8. Classe Interna SecureRequest

Encapsula os dados da requisição segura:

java
```
class SecureRequest {
    private String message;
    private long timestamp;

    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
    public long getTimestamp() { return timestamp; }
    public void setTimestamp(long timestamp) { this.timestamp = timestamp; }
}
```

Detalhes da Classe

- **Propriedades**: Contém as propriedades message e timestamp, representando a mensagem a ser validada e o tempo em que a requisição foi enviada.

3. Classe SecurityConfig

A classe SecurityConfig define as configurações de segurança para a API.

3.1. Anotações

- **@EnableWebSecurity**: Habilita a configuração de segurança do Spring, permitindo a personalização das regras de segurança.
- **@Configuration**: Indica que esta classe contém configurações específicas para o contexto de aplicação.

3.2. Método securityFilterChain

Configura a cadeia de filtros de segurança:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests().antMatchers("/api/token/login").permitAll()
        .anyRequest().authenticated()
        .and().httpBasic();

    return http.build();
}
```

Detalhes do Método

- **Desativação de CSRF**: O método desativa a proteção CSRF para simplificar os testes (não recomendado em produção).
- **Permissão de Acesso**:
  - **antMatchers("/api/token/login").permitAll()**: Permite acesso público ao endpoint de login.
  - **anyRequest().authenticated()**: Exige autenticação para todos os outros endpoints.
- **Autenticação Básica**: A configuração também habilita a autenticação básica HTTP.

4. Considerações

A implementação apresentada fornece uma base sólida para uma API segura utilizando HMAC e JWT em Spring.

Para garantir a segurança e a eficiência da aplicação, considere as seguintes recomendações:

1. **Segurança da Chave Secreta**: Armazene a chave secreta em um local seguro, como um serviço de gerenciamento de segredos.
2. **Persistência de Usuários**: Implemente autenticação contra um banco de dados em vez de usar credenciais hardcoded.
3. **Validação de Expiração**: Adicione lógica para lidar com tokens expirados e renovação de tokens conforme necessário.
4. **Tratamento de Exceções**: Implemente um tratamento de exceções robusto para fornecer mensagens de erro apropriadas e registrar eventos de falhas.

Este documento serve como um guia técnico para a implementação e segurança de APIs utilizando HMAC e JWT em Spring.

EducaCiência FastCode para comunidade