Aula 15: Implementando a Geração de Tokens JWT

September 18, 2025

1 Introdução

Na Aula 14, adicionamos a biblioteca Auth0 (`java-jwt`) ao projeto para suportar a geração de tokens JWT. Nesta aula, criaremos a classe `TokenService` para isolar a lógica de geração de tokens e atualizaremos o `AutenticacaoController` para retornar o token JWT na resposta do endpoint `/login`. Configuraremos o token com o algoritmo HMAC256 e uma validade de 2 horas. As alterações serão feitas no `Visual Studio Code` ou sua IDE preferida, com testes no Insomnia.

2 Criando a Classe `TokenService`

No pacote `med.voll.api.infra.security`, criamos a classe `TokenService` para gerenciar a geração de tokens JWT. O código é:

```
package med.voll.api.infra.security;

import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.exceptions.JWTCreationException;
import med.voll.api.domain.usuario.Usuario;
import org.springframework.stereotype.Service;
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneOffset;

@Service
public class TokenService {

    public String gerarToken(Usuario usuario) {
        try {
            var algoritmo = Algorithm.HMAC256("12345678");
            return JWT.create()
                    .withIssuer("API Voll.med")
                    .withSubject(usuario.getLogin())
                    .withExpiresAt(dataExpiracao())
                    .sign(algoritmo);
        } catch (JWTCreationException exception) {
            throw new RuntimeException("Erro ao gerar token JWT", exception);
        }
    }

    private Instant dataExpiracao() {
        return LocalDateTime.now().plusHours(2).toInstant(ZoneOffset.of("-03:00"));
    }
}
```

2.1 Explicação do Código

- `@Service`: Marca a classe como um serviço do Spring, permitindo injeção de dependência.
- `gerarToken(Usuario usuario)`: Gera um token JWT para o usuário autenticado.
- `Algorithm.HMAC256("12345678")`: Usa o algoritmo HMAC256 com uma chave secreta (neste caso, 12345678, que será ocultada em aulas futuras).
- `JWT.create()`: Configura o token com:
  - `withIssuer("API Voll.med")`: Define o emissor do token.
  - `withSubject(usuario.getLogin())`: Define o login do usuário como sujeito do token.
  - `withExpiresAt(dataExpiracao())`: Define a validade do token (2 horas).
  - `sign(algoritmo)`: Assina o token com o algoritmo HMAC256.
- `dataExpiracao()`: Calcula a data de expiração (2 horas a partir do momento atual, ajustada para o fuso horário -03:00).
- `JWTCreationException`: Captura erros na geração do token e lança uma `RuntimeException`.

3 Atualizando o `AutenticacaoController`

No `AutenticacaoController` (pacote `med.voll.api.controller`), injetamos o `TokenService` e atualizamos o método `efetuarLogin` para retornar o token JWT:

```
1  package med.voll.api.controller;
2
3  import jakarta.validation.Valid;
4  import med.voll.api.domain.usuario.DadosAutenticacao;
5  import med.voll.api.domain.usuario.Usuario;
6  import med.voll.api.infra.security.TokenService;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.http.ResponseEntity;
9  import org.springframework.security.authentication.AuthenticationManager;
10 import
       org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import org.springframework.web.bind.annotation.RequestMapping;
14 import org.springframework.web.bind.annotation.RestController;
15
16 @RestController
17 @RequestMapping("/login")
18 public class AutenticacaoController {
19
20     @Autowired
21     private AuthenticationManager manager;
22
23     @Autowired
24     private TokenService tokenService;
25
26     @PostMapping
27     public ResponseEntity efetuarLogin(@RequestBody @Valid DadosAutenticacao
           dados) {
28         var token = new UsernamePasswordAuthenticationToken(dados.login(),
               dados.senha());
29         var authentication = manager.authenticate(token);
30         return ResponseEntity.ok(tokenService.gerarToken((Usuario)
               authentication.getPrincipal()));
31     }
32 }
```

3.1 Explicação do Código

- `@Autowired private TokenService tokenService`: Injeta o serviço de geração de tokens.
- `(Usuario) authentication.getPrincipal()`: Obtém o objeto `Usuario` autenticado.
- `tokenService.gerarToken(...)`: Gera o token JWT usando o `TokenService`.
- `ResponseEntity.ok(...)`: Retorna o token com status 200 (OK).

4 Testando o Endpoint de Login

No Insomnia, testamos a requisição `POST http://localhost:8080/login` com o JSON:

```
{
    "login": "usuario@voll.med",
    "senha": "123456"
}
```

4.1 Resultado

- **Sucesso**: Retorna código 200 (OK) com o token JWT no corpo da resposta, no formato de uma string (ex.: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...`).

5 Próximos Passos

Com o token JWT sendo gerado e retornado, o próximo passo é implementar a validação do token nas requisições subsequentes, configurando o `Spring Security` para proteger os endpoints da API. Na próxima aula, criaremos um filtro para validar tokens JWT. Continue praticando no `Visual Studio Code` ou sua IDE preferida!

6 Dica do Professor

- **Aprofunde-se em JWT**: Consulte a documentação da biblioteca `java-jwt` (`https://github.com/auth0/java-jwt`) e explore exemplos de configuração de tokens.
- **Comunidade no X**: Siga perfis como @JWTExpert e @SpringAuth no X para dicas sobre tokens JWT e segurança em APIs.
- **Pratique**: No Insomnia, copie o token retornado e use ferramentas como `https://jwt.io` para decodificá-lo e verificar os campos `iss` (issuer), `sub` (subject) e `exp` (expiration).