SPRING

SPRING SECURITY (JWT)

FABIO TADASHI MIYASATO

3

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

3 SPRING SECURITY

O Spring Security é um framework de customização de autenticação e autorização, padrão para aplicações Spring.

Dentre as principais funcionalidades, destacam-se a proteção contra diversos ataques como session fixation, clickjacking e CSRF; além de integração com API Servlet e, claro, total otimização para aplicações Spring.

3.1 Autenticação x Autorização

Antes de iniciar a implementação do Spring Security, é importante conhecer os conceitos e a diferença entre autenticação e autorização. No contexto de uma aplicação Rest, a autenticação define se uma requisição pode ou não acessar a aplicação. A forma como o cliente autentica pode variar bastante. Neste capítulo serão implementados exemplos usando Basic Auth e JWT. Caso o cliente não esteja corretamente autenticado, as requisições devem ser retornadas com o HTTP Code 401 (Unauthorized).

A autorização indica que o cliente tem permissão para acessar a aplicação, ou seja, tem uma autenticação válida, mas não tem permissão para acessar determinada funcionalidade como um recurso específico ou tem apenas permissão de leitura — não de cadastrar um novo item. A forma mais comum de implementação (e que será utilizada neste capítulo) é o uso de *roles* (papéis).

3.2 Configuração Inicial

Este módulo pode ser implementado em qualquer aplicação Spring. Para facilitar, será utilizada a aplicação implementada do Capítulo 1 (Spring Framework), disponível no repositório https://github.com/fabiotadashi/spring/tree/modulo1.

A aplicação é uma API REST que trata do recurso “Livro”. Ela utiliza banco de dados embarcado H2 e possui os métodos básicos (CRUD) disponibilizados por endpoints.

Para acrescentar a dependência do Spring Security, basta adicionar a seguinte linha de declaração no arquivo build.gradle.

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```

Código-fonte 3.1 – Inclusão da dependência do Spring Security
Fonte: Elaborado pelo autor (2021)

O simples ato de incluir a dependência citada já protege os endpoints. Agora não é mais possível acessar o recurso GET => http://localhost:8088/livros. O resultado será uma resposta com erro HTTP 401.

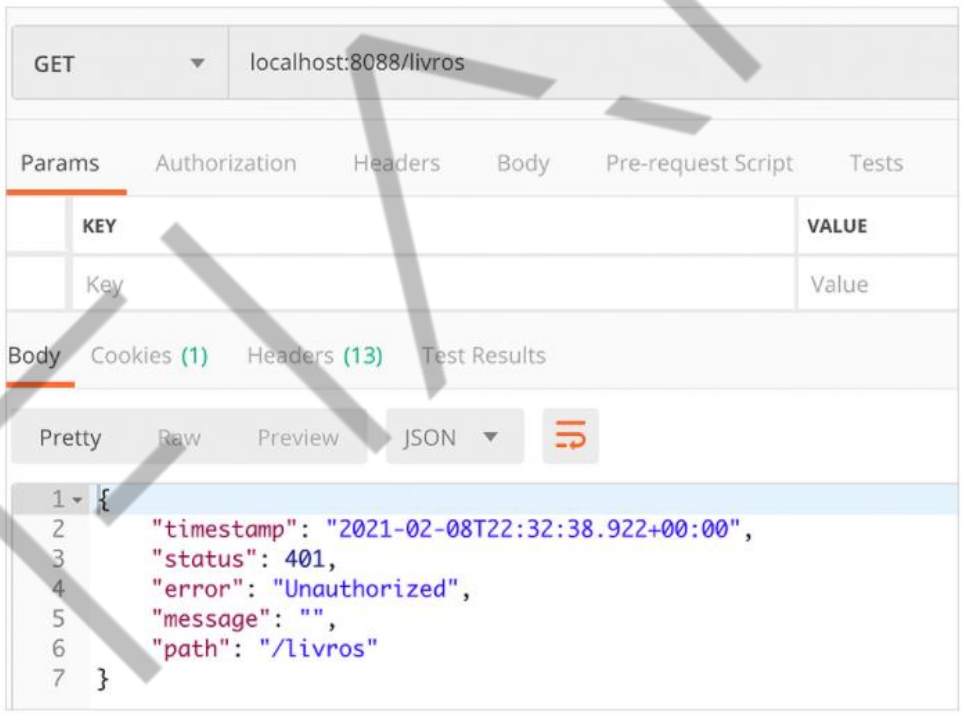

Figura 3.1 – Teste de execução (GET) com segurança habilitada
Fonte: Elaborado pelo autor (2021)

Para acessar o recurso novamente, pode-se observar o log da aplicação e procurar pela mensagem que contém a senha de acesso, conforme a figura “Log da aplicação com senha gerada automaticamente”.

Figura 3.2 – Log da aplicação com senha gerada automaticamente
Fonte: Elaborado pelo autor (2021)

O Spring gera a senha e um usuário padrão denominado “user”. A forma mais simples de testar uma API (aplicação) que tenha autenticação é por meio de um programa API Cliente como o Postman (https://www.postman.com).

No Postman, após selecionar o método GET e inserir a URL da aplicação (http://localhost:8088/livros), clique na aba “Authorization” e selecione “Basic Auth”. No usuário, digite “user”; e, em password, a senha gerada pelo Spring.

Ao fazer novamente a requisição, o HTTP Code retornado é 200, indicando sucesso na requisição, conforme a imagem “Configuração de parâmetros de autorização no Postman”.

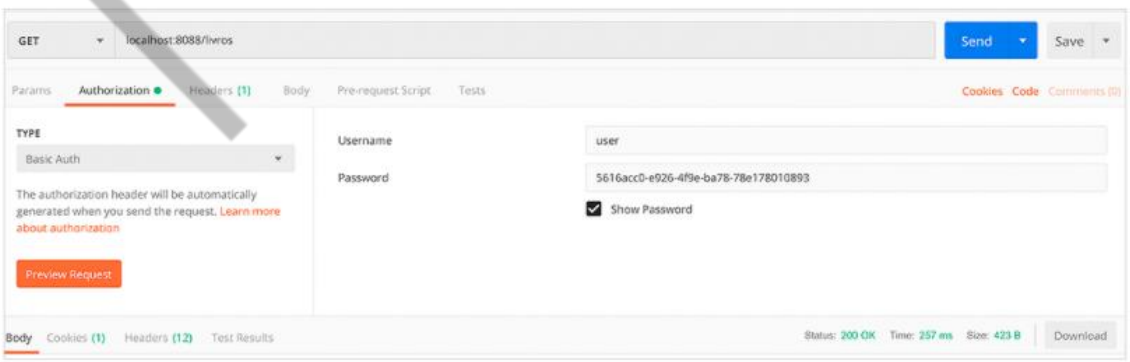

Figura 3.3 – Configuração de parâmetros de autorização no Postman
Fonte: Elaborado pelo autor (2021)

3.3 Basic Auth

A primeira forma de configurar a autenticação é a *Basic Auth* ou autenticação básica. Esse modelo segue a especificação RFC 7617 e funciona da seguinte forma: o cliente deve incluir, na requisição, o usuário e a senha dentro do Header "Authorization", concatenados com ":", utilizando o *encoding Base64*.

Esse modelo é bastante simples de configurar no Spring Security e atende à maioria das necessidades de uma aplicação básica. E veremos, com detalhes, ainda neste capítulo, a implementação usando JWT.

Além do usuário padrão, o Spring suporta a configuração de usuários e senhas no arquivo "application.yml".

```
spring:

  security:

   user:

     password: senha123

     name: admin

           roles: ADMIN
```

Código-fonte 3.2 – Configuração de credenciais de acesso no "application.yml"
Fonte: Elaborado pelo autor (2021)

Agora é possível fazer a requisição da listagem de livros com esse novo usuário:

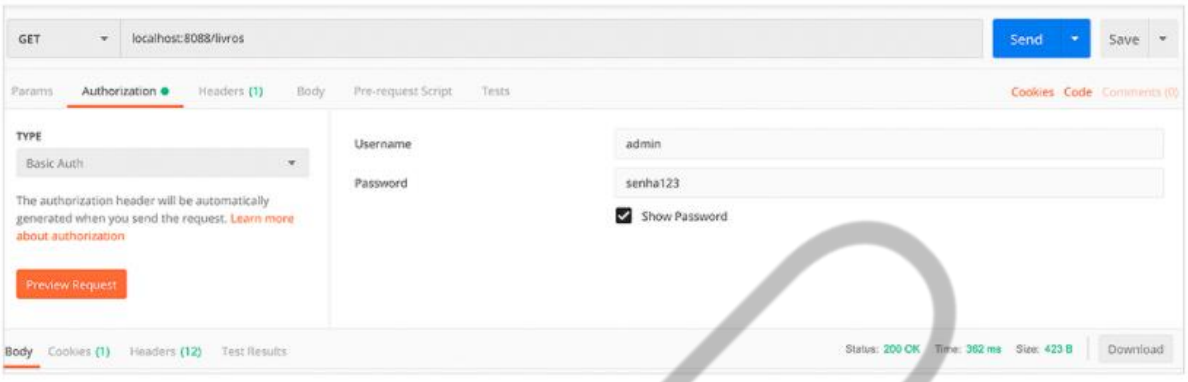

Figura 3.4 – Execução da requisição usando as credenciais de segurança configuradas
Fonte: Elaborado pelo autor (2021)

3.4 Configuração customizada

Apesar das facilidades oferecidas, o grande ganho de usar o Spring está na facilidade de customização das configurações. Para customizar a configuração, o primeiro passo é desabilitar a autoconfiguração do Spring Security, na classe principal de aplicação do Spring:

```
@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })
```

Código-fonte 3.3 – Desativação da segurança do Spring Security
Fonte: Elaborado pelo autor (2021)

Dessa forma, toda a segurança testada anteriormente estará desabilitada. Para criar uma configuração customizada será necessária uma nova classe dentro do pacote “configuration”, conforme o código-fonte “Estrutura da classe SecurityConfig”.

```
@Configuration

@EnableGlobalMethodSecurity(securedEnabled = true)

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter{
```

```
}
```

Código-fonte 3.4 – Estrutura da classe SecurityConfig
Fonte: Elaborado pelo autor (2021)

A classe disponibiliza alguns métodos que podem ser sobrescritos e customizados. Existem dois métodos chamados “configure” que recebem parâmetros diferentes. O método “configure” que recebe de parâmetro AuthenticationManagerBuilder gerencia a autenticação, e o que recebe de parâmetro HttpSecurity gerencia a autorização.

Para autenticação, o parâmetro AuthenticationManagerBuilder oferece várias formas. É nesse ponto que fica a definição se os usuários serão buscados no banco de dados, entre outras características. Para esta versão, vamos criar os usuários e autenticá-los apenas em memória.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
   auth.inMemoryAuthentication()
           .withUser("admin").password("{noop}admin").roles("ADMIN",
)
           .and()
           .withUser("user").password("{noop}user").roles("USER");
}
```

Código-fonte 3.5 – Método “configure” que manipula a autenticação
Fonte: Elaborado pelo autor (2021)

Com o código anterior, é possível acessar a aplicação com dois usuários que tenham senhas e roles (papéis) distintos. No caso da senha, o “{noop}” é importante, pois o Spring gerencia essas senhas de forma criptografada.

A autorização dos endpoints é configurada no método “configure”, que recebe o HttpSecurity como parâmetro. Nesse método também é possível habilitar e desabilitar diferentes configurações de segurança.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http
            .httpBasic()
            .and()
            .authorizeRequests()
            .antMatchers(HttpMethod.POST, "/livros").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .csrf().disable()
            .formLogin().disable();
}
```

Código-fonte 3.6 – Método “configure” que manipula a autorização
Fonte: Elaborado pelo autor (2021)

A autorização é criada pelo método “authorizeRequests()” por meio do “antMatcher”, com o método Http e o recurso. No código “Método “configure”, que manipula a autorização”, o acesso POST (http://localhost:8088/livros) está restrito aos usuários que possuem a role “ADMIN”. Ao tentar fazer o cadastro de um novo livro com o usuário “user”, o retorno é um erro com HTTP Code 403 (*Forbidden*).

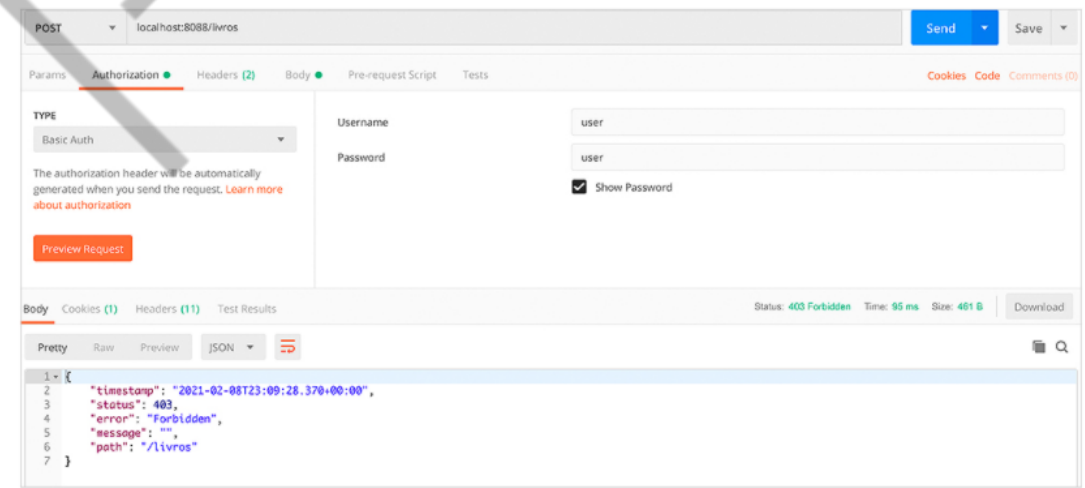

Figura 3.5 – Simulação de erro de autorização usando o usuário “user”
Fonte: Elaborado pelo autor (2021)

Ao tentar fazer essa mesma requisição com o usuário admin, o retorno é de sucesso HTTP Code 200 (OK).

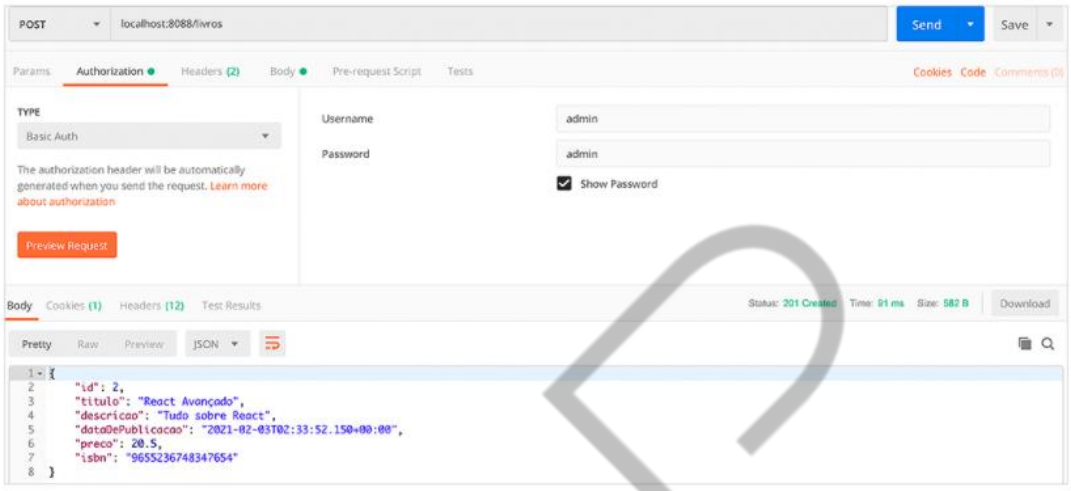

Figura 3.6 – Simulação de sucesso na requisição usando o usuário “admin”
Fonte: Elaborado pelo autor (2021)

3.5 JSON Web Token

Uma das formas mais difundidas no mercado é autenticação por meio de Json Web Token, ou simplesmente JWT (https://jwt.io). Trata-se de um token aberto, seguindo a RFC 7519 que permite o gerenciamento de autenticação seguro entre cliente e servidor. Assim como o Basic Auth, o token JWT é enviado no Header Authorization da requisição.

O token é dividido em 3 partes:

- Header: Indica o tipo de token e o algoritmo de hash utilizado.
- Payload: Os dados do usuário, quem gerou o token, tempo de expiração, entre outros dados.
- Assinatura: Valida que o token não foi alterado.

3.5.1 Preparação para implementação

O usuário será armazenado no banco de dados. Então, antes de implementar o JWT, é preciso criar a entidade, o script de migração e o repositório de usuários.

A entidade:

```
@Entity
@Table(name = "TB_USER")
@EntityListeners(AuditingEntityListener.class)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column
    private String username;
    @Column
    private String password;

  // getters e setters
}
```

Código-fonte 3.7 – Estrutura da entidade User
Fonte: Elaborado pelo autor (2021)

O script de migração é criado na pasta db/migrations no classpath e, seguindo o padrão de nomenclatura, deve ser nomeado para “V1_001__create_table_users.sql”. Para gerar a tabela, deve-se utilizar o script a seguir.

```
create table TB_USER (
    id bigint not null auto_increment,
    username varchar(100) not null,
    password varchar(100) not null,
    primary key (id)
```

```
)
```

Código-fonte 3.8 – Script de geração da tabela
Fonte: Elaborado pelo autor (2021)

No *package* “repository”, devem ser criados o repositório de acesso aos usuários e o método para buscar o usuário por nome.

```
public interface UserRepository extends JpaRepository<User, Long> {

   Optional<User> findFirstByUsername(String username);

}
```

Código-fonte 3.9 – Interface UserRepository
Fonte: Elaborado pelo autor (2021)

3.5.2 Guia de implementação

Para implementar a autenticação e a autorização via JWT com banco de dados, será necessário criar 05 componentes que serão detalhados mais adiante. De forma resumida, são:

- TokenUtil: Centraliza os métodos de gerenciamento do token.
- AuthenticationEntryPoint: Trata as exceções devolvendo erro 401 Unauthorized.
- RequestFilter: Valida os tokens na requests.
- UserDetails: Trata os dados do usuário.
- PasswordEncoder: Criptografa a senha do usuário.

O primeiro passo é incluir, no arquivo dependencies.gradle, a dependência do jwt.

```
implementation 'io.jsonwebtoken:jjwt:0.9.1'
```

Código-fonte 3.10 – Inclusão da dependência do JWT
Fonte: Elaborado pelo autor (2021)

3.5.3 Token Util

Esse componente tem como função gerar e ler o token. Vamos criar a classe JwtTokenUtil no *package* “security” e criar duas propriedades que serão recebidas via parametrização no arquivo “application.yml”. Essa classe deve ser anotada com @Component, a fim de ser utilizada por meio de injeção de dependência em outras classes.

```
@Component
public class JwtTokenUtil {

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

   @Value("${jwt.expire}")
   private int expire;
}
```

Código-fonte 3.11 – Estrutura da classe “JwtTokenUtil”
Fonte: Elaborado pelo autor (2021)

As propriedades a seguir representam um secret (ou salt) para manipulação do token e o tempo de expiração em minutos. A parametrização dessas propriedades deve ser feita no “application.yml”.

```
jwt:
 secret: f1aPs3cr37
```

```
      expire: 5
```

Código-fonte 3.12 – Configuração do secret/salt para manipular o token
Fonte: Elaborado pelo autor (2021)

Depois, deve ser criado o método generateToken que recebe como parâmetro o nome do usuário, que é configurado no payload do JWT por meio do método “setSubject”. É neste ponto que o tempo de expiração do token deve ser configurado por meio da propriedade “jwt.expire”. Para facilitar, pode-se criar o método privado “getFromLocalDate”. A assinatura do token é gerada utilizando o secret e o algoritmo HS512.

```
public String generateToken(String username){
 Map<String, Object> claims = new HashMap<>(); // Roles
 Date dataCriacao = getFromLocalDate(LocalDateTime.now());
 Date dataExpiracao =
      getFromLocalDate(LocalDateTime.now().plusMinutes(expire));
 return Jwts.builder()
           .setClaims(claims)
           .setIssuedAt(dataCriacao)
           .setExpiration(dataExpiracao)
           .setSubject(username)
           .signWith(SignatureAlgorithm.HS512, secret)
           .compact();
}
 private Date getFromLocalDate(LocalDateTime now) {
    return Date.from(now.toInstant(OffsetDateTime.now().getOffset()));
 }
```

Código-fonte 3.13 – Configuração de expiração do token
Fonte: Elaborado pelo autor (2021)

Em seguida, deve ser criado o método “getUserNameFromToken” que recebe como parâmetro o token. Tais métodos serão bastante úteis para as classes que precisam ler e gerar os tokens.

```
    public String getUsernameFromToken(String token){
        Claims claims = Jwts.parser()
                .setSigningKey(secret)
                .parseClaimsJws(token.replace("Bearer ", ""))
                .getBody();
        return claims.getSubject();
    }
```

Código-fonte 3.14 – Estrutura do método “getUsernameFromToken”
Fonte: Elaborado pelo autor (2021)

3.5.4 JwtAuthenticationEntryPoint

A classe “JwtAuthenticationEntryPoint” tem a responsabilidade de incluir o HTTP code 401 (Unauthorized) nas requisições que lançam AuthenticationException. Deve-se criar a classe no pacote “security”, anotar com @Component e implementar a interface AuthenticationEntryPoint, que tem um único método denominado “commence”.

```
@Component
public class JwtAuthenticationEntrypoint
            implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
```

```
                    AuthenticationException authException)
  throws IOException {
     response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
ten.invalido");
     }
  }
```

Código-fonte 3.15 – Estrutura do método “commence”
Fonte: Elaborado pelo autor (2021)

3.5.5 JwtUserDetailService

A classe “JwtUserDetailService” trata os dados do usuário e faz a ligação entre o Spring Security e o banco de dados que está sendo utilizado. Deve-se criar a classe JwtUserDetailService dentro do pacote “security” e implementar a interface “UserDetailsService”. E, na sequência, configura-se o repositório “UserRepository” por meio da injeção de dependência por construtor.

```
@Component
public class JwtUserDetailService implements UserDetailsService {
   private final UserRepository userRepository;
   public JwtUserDetailService(UserRepository userRepository) {
       this.userRepository = userRepository;
   }
}
```

Código-fonte 3.16 – Estrutura da classe JwtUserDetailService
Fonte: Elaborado pelo autor (2021)

O próximo passo é sobrescrever o método “loadByUserName” da interface UserDetailsService. Esse método recebe o nome do usuário e retorna uma instância da interface UserDetails, que faz parte do Spring Security.

```
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findFirstByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not
found"));

        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                new ArrayList<>() // Roles
        );
    }
```

Código-fonte 3.17 – Sobrescrita do método “loadUserByUsername”
Fonte: Elaborado pelo autor (2021)

3.5.6 JwtRequestFilter

O componente “JwtRequestFilter” é responsável pela validação do token. Agora, seguimos criando a classe “JwtRequestFilter” no pacote “security”. A classe deve ser anotada com “@Component” e deve estender a classe “OncePerRequestFilter”. Nessa classe devem ser incluídas as dependências de dois componentes criados anteriormente: JwtUserDetailService e JwtTokenUtil.

```
    @Component
    public class JwtFilter extends OncePerRequestFilter {
        private final JwtUserDetailService jwtUserDetailService;
        private final JwtTokenUtil jwtTokenUtil;
```

```
        public JwtFilter(JwtUserDetailService jwtUserDetailService,
                         JwtTokenUtil jwtTokenUtil) {
            this.jwtUserDetailService = jwtUserDetailService;
            this.jwtTokenUtil = jwtTokenUtil;
        }
    }
```

Código-fonte 3.18 – Estrutura da classe JwtFilter
Fonte: Elaborado pelo autor (2021)

Em seguida, deve-se sobrescrever o método “doInternalFilter” que irá extrair o nome do usuário do token e incluir os *UserDetails* no contexto de segurança do Spring (SecurityContextHolder).

```
    @Override
    protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain)
    throws ServletException, IOException {
    final String authorizationHeaderToken =
request.getHeader("Authorization");
    String username = null;

    if (authorizationHeaderToken != null &&
authorizationHeaderToken.startsWith("Bearer ")) {
    try {
    username = jwtTokenUtil.getUsernameFromToken(authorizationHeaderToken);
    } catch (IllegalArgumentException illegal) {
    logger.info(illegal.getMessage());
    } catch (ExpiredJwtException expired) {
    logger.info(expired.getMessage());
    }
```

```
        } else {
        logger.warn("Token null ou fora do padrao Bearer");
        }

        if (username != null &&
        SecurityContextHolder.getContext().getAuthentication() == null) {
        UserDetails userDetails =
jwtUserDetailService.loadUserByUsername(username);
        UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(userDetails,
        null,
        userDetails.getAuthorities());// Role

        authenticationToken.setDetails(
        new WebAuthenticationDetailsSource().buildDetails(request)
        );
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        }

        filterChain.doFilter(request, response);
        }
```

Código-fonte 3.19 – Estrutura do método “doFilterInternal”
Fonte: Elaborado pelo autor (2021)

3.5.7 PasswordEncoder

A senha do usuário nunca deve ser gravada em formato *plain text* no banco de dados. Para gerenciar a criptografia, deve ser criado o componente PasswordEncoder por meio de um @Bean, na classe AuthConfig. O componente deve ser colocado no pacote “configuration”.

```
        @Configuration
```

```
public class AuthConfig {

    @Bean

    public PasswordEncoder passwordEncoder(){

        return new BCryptPasswordEncoder();

    }

}
```

Código-fonte 3.20 – Estrutura da classe “AuthConfig”
Fonte: Elaborado pelo autor (2021)

3.5.8 Integrando os componentes

Agora, devemos alterar a classe SecurityConfig excluindo a customização de Basic Auth e incluindo as dependências dos componentes que foram criados.

```
    private final JwtAuthenticationEntrypoint
jwtAuthenticationEntrypoint;

    private final JwtUserDetailService jwtUserDetailService;

    private final JwtFilter jwtFilter;

    private final PasswordEncoder passwordEncoder;


    public SecurityConfig(JwtAuthenticationEntrypoint
jwtAuthenticationEntrypoint,
                          JwtUserDetailService jwtUserDetailService,
                          JwtFilter jwtFilter,
                          PasswordEncoder passwordEncoder){
        this.jwtAuthenticationEntrypoint = jwtAuthenticationEntrypoint;
        this.jwtUserDetailService = jwtUserDetailService;
        this.jwtFilter = jwtFilter;
        this.passwordEncoder = passwordEncoder;
```

Código-fonte 3.21 – Desativação da autenticação básica e habilitação do JWT
Fonte: Elaborado pelo autor (2021)

Em seguida, deve-se transformar a classe AuthenticationManager (filha da classe WebSecurityConfigurerAdapter) em um Bean.

```
@Bean
@Override
public AuthenticationManager authenticationManager()
                                                throws Exception {
   return super.authenticationManager();
}
```

Código-fonte 3.22 – Estrutura parcial da classe “AuthenticationManager”
Fonte: Elaborado pelo autor (2021)

O método “configure” define a autenticação (recebe AuthenticationManagerBuilder como parâmetro) e agora deve utilizar a dependência de JwtUserDetailService e o PasswordEncoder.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
   auth.userDetailsService(jwtUserDetailService)
         .passwordEncoder(passwordEncoder);
}
```

Código-fonte 3.23 – Estrutura do método “configure” para autenticação
Fonte: Elaborado pelo autor (2021)

O método “configure” define também a autorização por meio de JwtAuthenticationEntryPoint e do filtro JwtFilter. É importante ressaltar que o endpoint “users” é liberado no método authorizeRequest. Esse endpoint será desenvolvido posteriormente para criação e login de usuários. Logo, não deve ser autenticado.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
```

```
          http.httpBasic()
                    .and()
                    .authorizeRequests()
                    .antMatchers("/users/**").permitAll()
                    .anyRequest().authenticated()
                    .and()
                    .exceptionHandling()
                     .authenticationEntryPoint(jwtAuthenticationEntrypoint)
                    .and()
                    .sessionManagement()
      .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                    .and()
.addFilterBefore(jwtFilter,UsernamePasswordAuthenticationFilter.class)
                    .csrf().disable()
                    .formLogin().disable();
      }
```

Código-fonte 3.24 – Estrutura do método “configure” para autorização
Fonte: Elaborado pelo autor (2021)

3.5.9 Criação e autenticação de usuários

Para criar e autenticar os usuários são necessários 04 objetos do tipo DTO (*Data Transfer Objects*).

1 - AuthDTO para receber o usuário e senha no endpoint de login:

```
      public class AuthDTO {
         privatemel String username;
         private String password;
         // getters e setters
      }
```

2 - JwtDTO para retornar o token no endpoint de login:

```
public class JwtDTO {

   private String token;

   // getter e setter

}
```

3 - UserCreateDTO para criação de um novo usuário:

```
public class UserCreateDTO {

   private String username;

   private String password;

   // getters e setters

}
```

4 - UserDTO para retornar o usuário criado:

```
public class UserDTO {

   private Long id;

   private String username;

   // getters e setters

}
```

A criação separada das classes DTO, além de aderência a boas práticas, é importante também para permitir o gerenciamento correto das assinaturas dos métodos. **É importante ressaltar que, para criar um usuário, por exemplo, a aplicação recebe o usuário e a senha, mas não o ID. E, ao retornar o usuário criado, é devolvido o ID e o nome do usuário, mas não a senha.**

Agora, deve ser criada a interface do serviço UserService no pacote “service”.

```
public interface UserService {

   UserDTO create(UserCreateDTO userCreateDTO);

   JwtDTO login(AuthDTO authDTO);

}
```

Código-fonte 3.25 – Estrutura da interface “UserService”
Fonte: Elaborado pelo autor (2021)

Na implementação da classe “UserServiceImpl” devem ser incluídas as dependências de JwtTokenUtil, AuthenticationManager, PasswordEncoder e UserRepository.

```
@Service
public class UserServiceImpl implements UserService {
   private final JwtTokenUtil jwtTokenUtil;
   private final AuthenticationManager authenticationManager;
   private PasswordEncoder passwordEncoder;
   private UserRepository userRepository;
   public UserServiceImpl(JwtTokenUtil jwtTokenUtil,
            AuthenticationManagerauthenticationManager,
                        PasswordEncoder passwordEncoder,
                        UserRepository userRepository){
      this.jwtTokenUtil = jwtTokenUtil;
      this.authenticationManager = authenticationManager;
      this.passwordEncoder = passwordEncoder;
      this.userRepository = userRepository;
   }
```

Código-fonte 3.26 – Estrutura da classe “UserServiceImpl”
Fonte: Elaborado pelo autor (2021)

Em seguida, devem ser implementados os métodos de login e criação de usuário.

```
        @Override
        public UserDTO create(UserCreateDTO userCreateDTO) {
            User user = new User();
            user.setUsername(userCreateDTO.getUsername());
user.setPassword(passwordEncoder.encode(userCreateDTO.getPassword()));
            User savedUser = userRepository.save(user);
            UserDTO userDTO = new UserDTO();
            userDTO.setId(savedUser.getId());
            userDTO.setUsername(savedUser.getUsername());
            return userDTO;
        }
        @Override
        public JwtDTO login(AuthDTO authDTO) {
            try{
                authenticationManager.authenticate(
                        new
UsernamePasswordAuthenticationToken(authDTO.getUsername(),
authDTO.getPassword())
                );
            } catch (AuthenticationException e) {
                throw new ResponseStatusException(HttpStatus.UNAUTHORIZED,
e.getMessage());
            }
            String token = jwtTokenUtil.generateToken(authDTO.getUsername());
            JwtDTO jwtDTO = new JwtDTO();
            jwtDTO.setToken(token);
            return jwtDTO;
        }
```

Código-fonte 3.27 – Estrutura do método “create”
Fonte: Elaborado pelo autor (2021)

Por fim, deve ser criado o *controller* UserController no pacote “controller”. Deve ser incluída também a dependência do componente UserService seguida da configuração das assinaturas.

```
@RestController
@RequestMapping("users")
public class UserController {
    private UserService userService;
    public UserController(UserService userService){
        this.userService = userService;
    }
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public UserDTO createUser(@RequestBody UserCreateDTO
userCreateDTO){
        return userService.create(userCreateDTO);
    }
    @PostMapping("login")
    public JwtDTO login(@RequestBody AuthDTO authDTO){
        return userService.login(authDTO);
    }
}
```

Código-fonte 3.28 – Estrutura da classe “UserController”
Fonte: Elaborado pelo autor (2021)

3.6 Criando usuários e chamando os métodos

Ao executar a aplicação, não temos usuários. Para criar um usuário, basta invocar o endpoint “/users” que não requer autenticação. O *body* é um objeto json com os campos *username* e *password*.

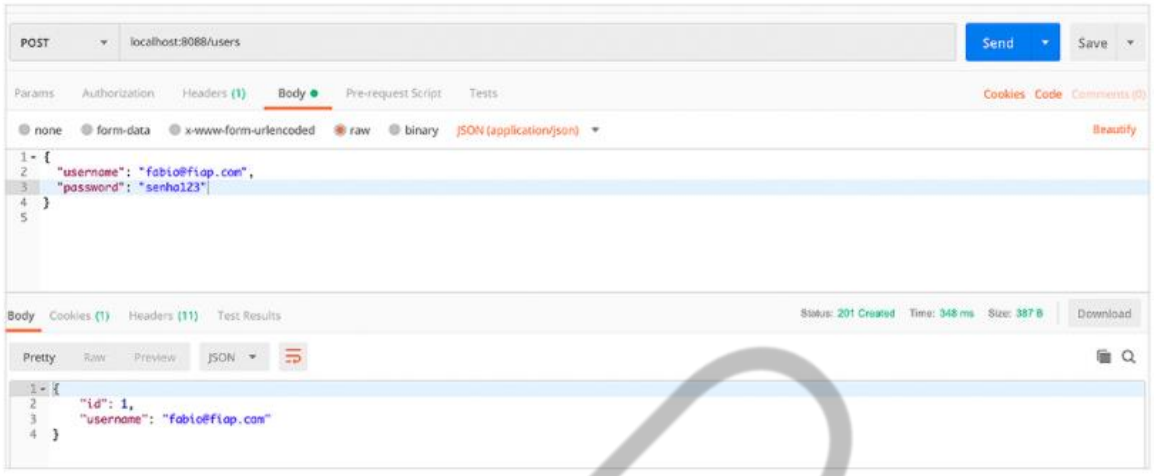

Figura 3.7 – Requisição para criação de usuário
Fonte: Elaborado pelo autor (2021)

O endpoint cria um usuário e retorna o HTTP code 201 (Created). Agora, o usuário e a senha podem ser utilizados no endpoint “/users/login” com o mesmo *payload* (body).

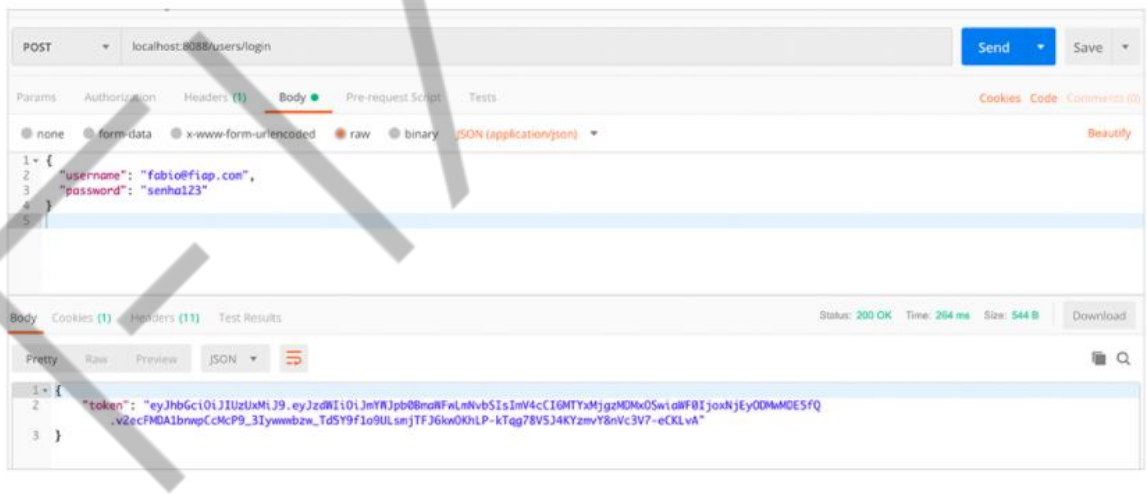

Figura 3.8 – Teste de requisição com o usuário criado anteriormente
Fonte: Elaborado pelo autor (2021)

Na parte inferior da Figura “Teste de requisição com o usuário criado anteriormente” podem ser observados o retorno HTTP code 200 (OK) e um objeto com o token de acesso.

Ao fazer uma chamada na lista de livros (GET => http://localhost:8088/livros), o retorno é o HTTP code 401 (Unauthorized). No postman, podemos acessar a aba

authorization e selecionar Bearer Token, padrão adotado na implementação JWT. No campo token deve ser inserido o valor “Bearer token_retornado_pelo_login”.

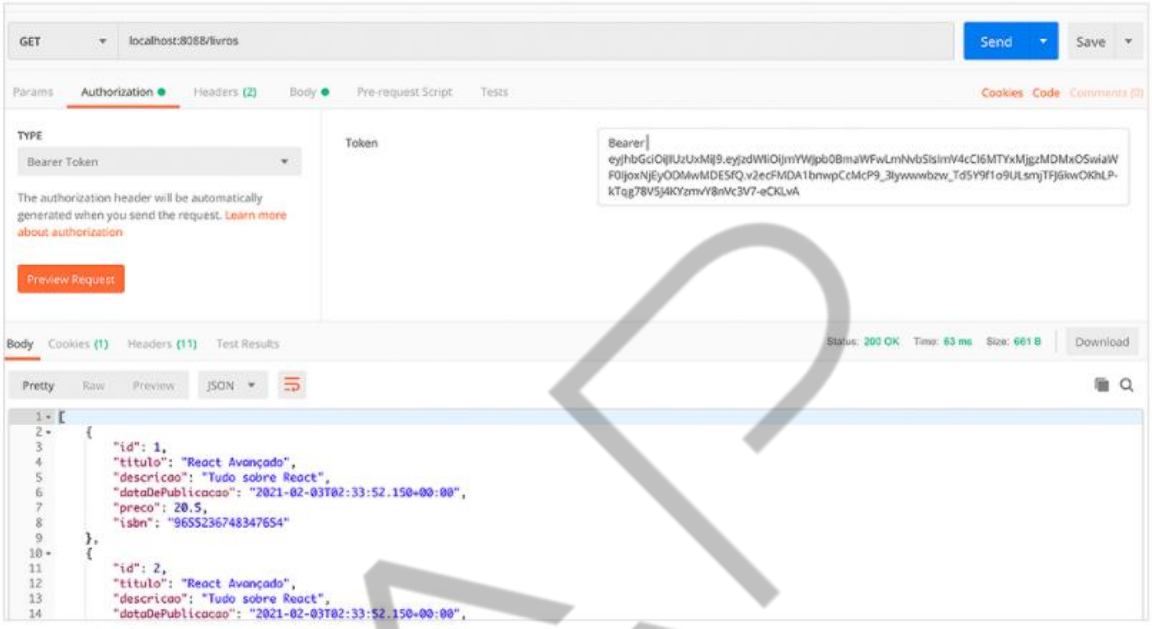

Figura 3.9 – Configuração do tipo de token que será usado na requisição
Fonte: Elaborado pelo autor (2021)

É possível visualizar os dados do token no site jwt.io. Porém, manipular os dados do token torna-o inválido.

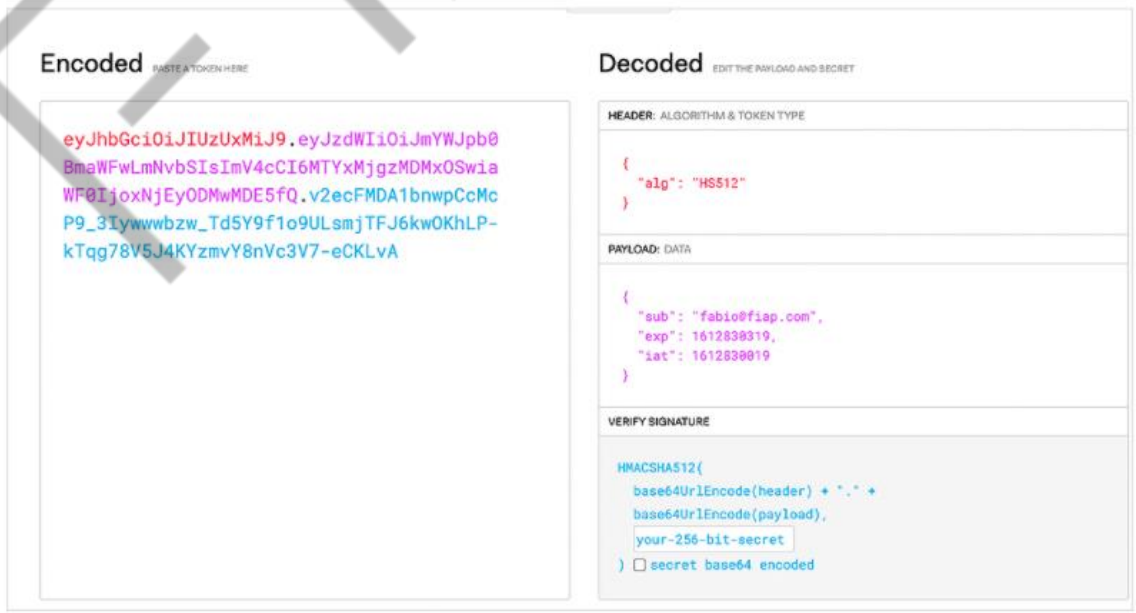

Figura 3.10 – Visualização do token no site jwt.io
Fonte: Elaborado pelo autor (2021)

Tanto a expiração quanto a alteração do token fazem com que o servidor retorne o HTTP *status code* 401 (Unauthorized) no processo de autenticação.

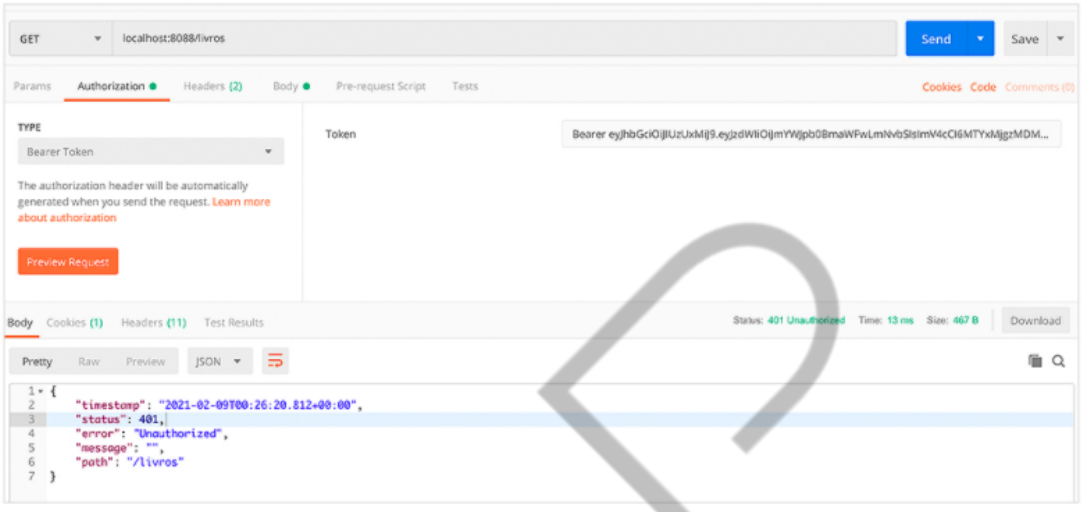

Figura 3.11 – Simulação de erro de autenticação (HTTP status code 401)
Fonte: Elaborado pelo autor (2021)

Conclusão

Neste capítulo foram implementadas duas formas de autenticação: Basic Auth e JWT, além de cobrir os conceitos básicos de autenticação e autorização em uma API Rest. A aplicação desenvolvida compreende a criação e o login de usuários, além de autenticação e expiração do token de maneira segura, utilizando o Spring Security.

O projeto desenvolvido, ao longo do capítulo, está publicado em https://github.com/fabiotadashi/spring/tree/modulo_security. Finalizamos, portanto, o conteúdo de Spring. Nas lives, abordaremos particularidades e boas práticas, além de outros exemplos.

Até breve e bons estudos!

REFERÊNCIAS

CARNELL, John. **Spring Microservices in Action**. First Edition. Manning, 2017.

Spring Framework Documentation. Disponível em: <https://docs.spring.io/spring-framework/docs/current/reference/html/>. Acesso em: 25 jan. 2021.

GLOSSÁRIO

RFC 7617	Acrônimo de Request for Comments. Trata-se de documentos técnicos desenvolvidos e mantidos pela IETF. IEFT é o acrônimo de Internet Engineering Task Force, uma grande comunidade internacional aberta de designers de rede, operadoras, fornecedores e pesquisadores preocupados com a evolução da arquitetura da Internet e seu bom funcionamento.
DTO	Data Transfer Object