Desenvolvimento para Servidores-II

Spring Security

Neste tópico abordaremos autenticação e autorização usando Spring Security

Prof. Ciro Cirne Trindade

Spring Security

- Spring Security é um framework poderoso e customizável para autenticação e controle de acesso em aplicações Java
- Baseado em JWT (JSON Web Tokens)
  - https://jwt.io/

JWT (1/4)

- Padrão aberto (RFC 7519) para a troca de mensagens seguras entre duas partes através de objetos JSON
- Características
  - Leve: usa JSON
  - Autocontido: tem todas as informações necessárias para seu processamento
  - Seguro: a informação pode ser verificada e é confiável porque ela é assinada digitalmente

JWT (2/4)

- Estrutura do JWT
  - 3 partes separadas por pontos (.)
    - **Header**: define o tipo do token e o algoritmo de criptografia
    - **Payload**: normalmente contém as informações do usuário autenticado
    - **Signature**: concatenação dos hashes de Header e Payload com uma chave secreta
  - Portanto um JWT se parece com o seguinte
    - xxxxx.yyyyyy.zzzzzz

JWT (3/4)

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

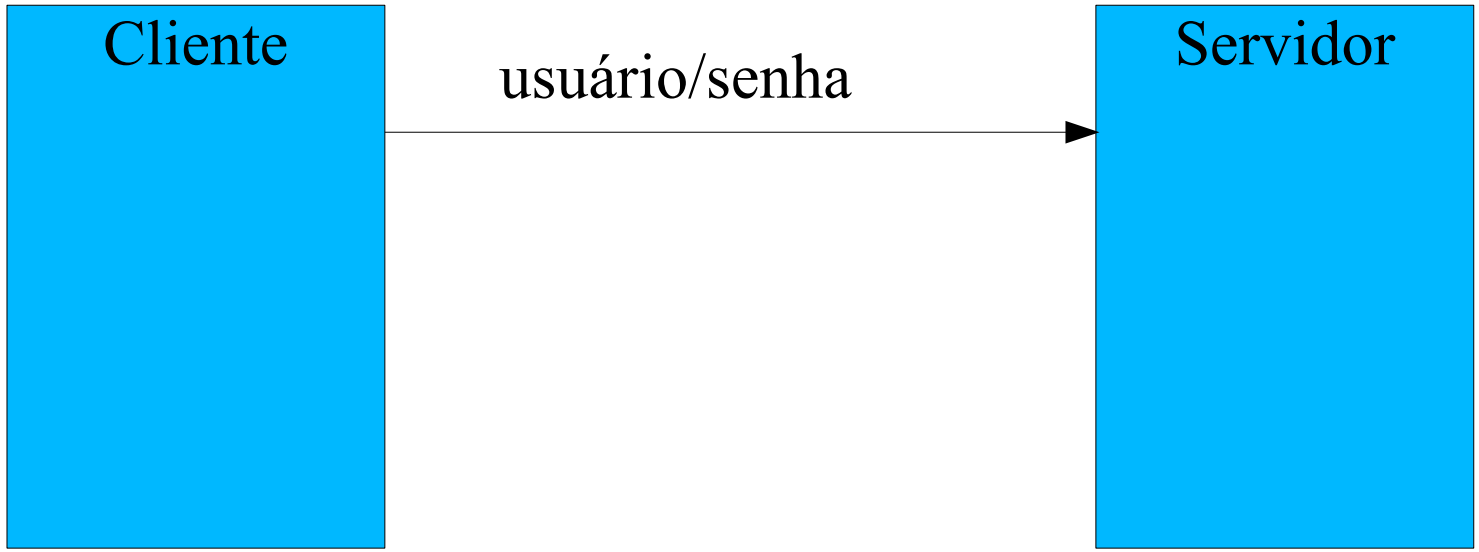
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

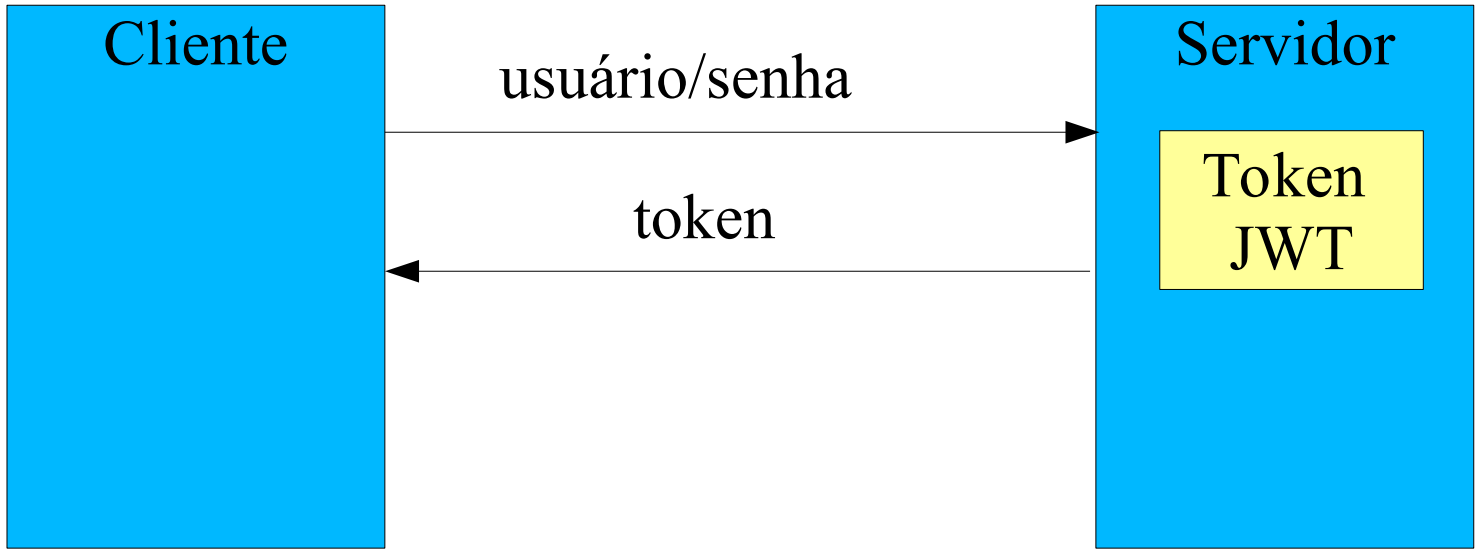
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

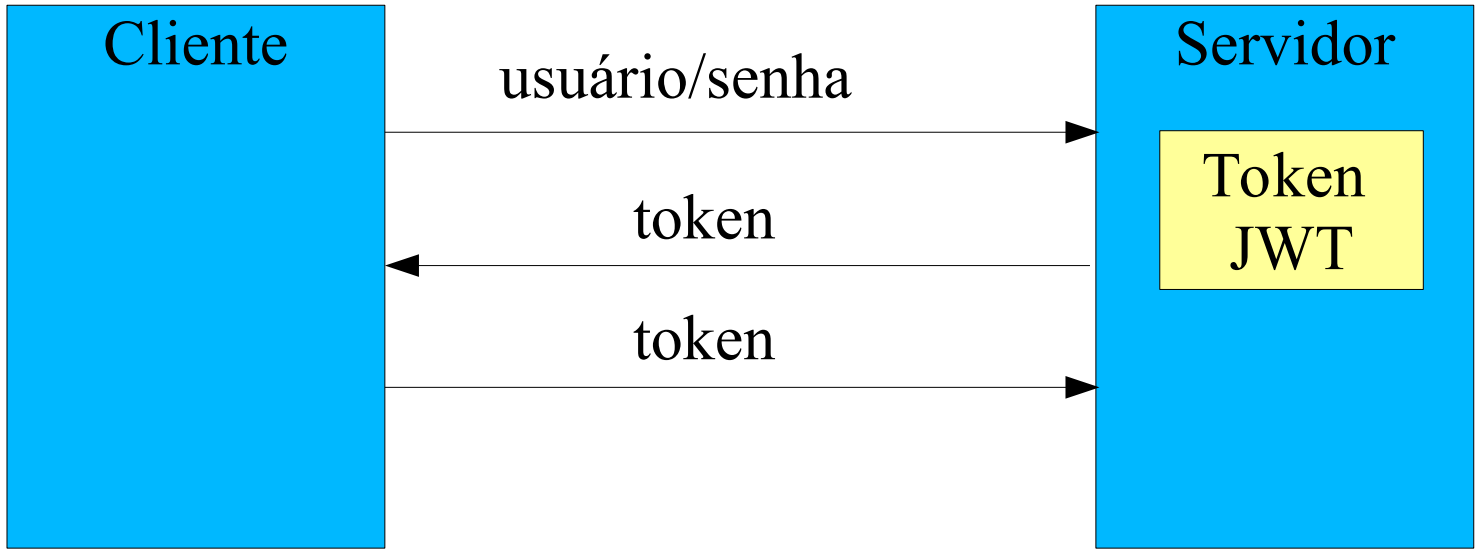
JWT (4/4)



JWT (4/4)



JWT (4/4)



Configurações iniciais (1/2)

- Incluir as dependências no pom.xml

```
<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>
```

A inclusão desta dependência já bloqueia as requisições

Configurações iniciais (2/2)

- Criar a classe de configuração para definir as configurações de segurança
  - Esta classe deve herdar de `WebSecurityConfigurerAdapter`
  - Define as configurações básicas das URL's que necessitam ou não de autenticação/autorização
  - Normalmente isso é feito sobrescrevendo o método `configure`

SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends
                                WebSecurityConfigurerAdapter {

    private static final String[] PUBLIC_MATCHERS = {
        "/categorias/**",
        "/pessoas_fisicas/**",
        "/pessoas_juridicas/**"
    };

    @Override
    protected void configure(HttpSecurity http) throws
                                                    Exception {
        http.authorizeRequests().antMatchers(PUBLIC_MATCHERS)
            .permitAll().anyRequest().authenticated();
    }
}
```

URLs liberadas

Configurações adicionais (1/2)

- Desabilitar proteção a CSRF (*Cross-Site Request Forgery*) e definir o sistema como *stateless*

```
http.csrf().disable();
http.sessionManagement().sessionCreationPolicy(
                              SessionCreationPolicy.STATELESS);
```

- Permitindo apenas requisições do tipo GET

```
http.authorizeRequests()
        .antMatchers(HttpMethod.GET, PUBLIC_MATCHERS)
        .permitAll().anyRequest().authenticated();
```

Default

Configurações adicionais (2/2)

- CORS (*Cross-Origin Resource Sharing*)
  - Permite que a aplicação compartilhe recursos de múltiplas fontes

```
@Override
protected void configure(HttpSecurity http) throws Exception {
   http.cors().and().csrf().disable();
   http.authorizeRequests().antMatchers(HttpMethod.GET,
        PUBLIC_MATCHERS).permitAll().anyRequest().authenticated();
   http.sessionManagement().sessionCreationPolicy(
                                SessionCreationPolicy.STATELESS);
}

@Bean
CorsConfigurationSource corsConfigurationSource() {
   final UrlBasedCorsConfigurationSource source = new
                                UrlBasedCorsConfigurationSource();
   source.registerCorsConfiguration("/**", new
                  CorsConfiguration().applyPermitDefaultValues());
   return source;
}
```

Adicionando login, senha e perfis a classe Cliente

- Para fazer a autenticação, vamos precisar incluir os atributos login, senha e perfis na classe `Cliente`
- Os perfis do usuário serão definidos através de uma enumeração

TipoPerfil.java

```
public enum TipoPerfil {
    ADMIN(1, "ROLE_ADMIN"),
    CLIENTE(2, "ROLE_CLIENTE");

    private Integer cod;
    private String descricao;

    private TipoPerfil(Integer cod, String descricao) {
        this.cod = cod;
        this.descricao = descricao;
    }

    public Integer getCod() { return cod; }

    public String getDescricao() { return descricao; }

    public static TipoPerfil toEnum(Integer cod) {
        if (cod == null) return null;
        for (TipoPerfil x : TipoPerfil.values()) {
            if (cod.equals(x.getCod())) return x;
        }
        throw new IllegalArgumentException("Código inválido: " + cod);
    }
}
```

Por padrão a descrição de um perfil no Spring Security deve começar por ROLE

Perfis como um `Set<Integer>`

- Para facilitar a persistência no banco de dados, o atributo `perfis` do `Cliente` será um `Set<Integer>` e não um `Set<TipoPerfil>`
- Entretanto, os métodos `getPerfis()` e `addPerfil()` tratam os perfis como `TipoPerfil`

Perfis

- Cliente.java

```
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "tb_perfil")
private Set<Integer> perfis = new HashSet<>();

public Set<TipoPerfil> getPerfis() {
    return perfis.stream()
     .map(x -> TipoPerfil.toEnum(x))
     .collect(Collectors.toSet());
}

public void addPerfil(TipoPerfil perfil) {
    this.perfis.add(perfil.getCod());
}
```

Relacionamento do tipo muitos para muitos

Adicionando login e senha ao Cliente

- Cliente.java

```
@Column(name = "nm_login", length = 80, unique = true)
private String login;

@Column(name = "nm_senha")
private String senha;

public String getLogin() { return login; }

public void setLogin(String login) {
   this.login = login;
}

@JsonIgnore
public String getSenha() { return senha; }

@JsonProperty
public void setSenha(String senha) {
   this.senha = senha;
}
```

Criptografando a senha nas classes de serviço

- Vamos criptografar a senha do cliente usando a classe `BcryptPasswordEncoder` em `PessoaFisicaService` e `PessoaJuridicaService`

```
@Autowired
private BCryptPasswordEncoder passwordEncoder;

@Override
public PessoaFisicaDTO create(PessoaFisicaDTO obj) {
    obj.setSenha(passwordEncoder.encode(obj.getSenha()));
    PessoaFisica pf = repository.save(mapper.toEntity(obj));
    return mapper.toDTO(pf);
}
```

Construtor de BcryptPasswordEncoder

- É necessário acrescentar um método produtor para `BcryptPasswordEncoder` na classe SecurityConfig

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Liberando o POST para PessoaFisica e PessoaJuridica

- SecurityConfig.java

```
private static final String[] PUBLIC_MATCHERS_POST = {
    "/pessoas_fisicas/**",
    "/pessoas_juridicas/**"
};

@Override
protected void configure(HttpSecurity http) throws Exception
{
    http.cors().and().csrf().disable();
    http.authorizeRequests()
        .antMatchers(HttpMethod.GET, PUBLIC_MATCHERS)
        .permitAll()
        .antMatchers(HttpMethod.POST, PUBLIC_MATCHERS_POST)
        .permitAll()
        .anyRequest().authenticated();
    http.sessionManagement().sessionCreationPolicy(
                            SessionCreationPolicy.STATELESS);
}
```

Implementando `UserDetails`

- Classe que contém as credenciais do usuário do Spring Security
- Armazena informações do usuário que posteriormente são encapsuladas em um objeto do tipo `Authentication`

UserDetailsImpl.java (1/2)

```
public class UserDetailsImpl implements UserDetails {
   private static final long serialVersionUID = 1L;

   private Long id;
   private String login;
   private String senha;
   private Collection<? extends GrantedAuthority> authorities;

   public UserDetailsImpl() { }

   public UserDetailsImpl(Long id, String login, String senha,
                          Set<TipoPerfil> perfis) {
      super();
      this.id = id;
      this.login = login;
      this.senha = senha;
      this.authorities = perfis.stream()
          .map(x -> new SimpleGrantedAuthority(x.getDescricao()))
          .collect(Collectors.toList());
   }

   public Long getId() { return id; }
```

UserDetailsImpl.java (2/2)

```
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
       return authorities;
    }

    @Override
    public String getPassword() { return senha; }

    @Override
    public String getUsername() { return login; }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }
}
```

Implementando `UserDetailsService`

- Classe que permite recuperar um usuário pelo username (login)
- Define apenas um método
  - `loadUserByUsername(String username)`
  - Localiza um usuário baseado no seu username (login)
  - Devolve um `UserDetails`

ClienteRepository.java

```
@Repository
public interface ClienteRepository extends
                            JpaRepository<Cliente, Long> {

    Cliente findByLogin(String login);

}
```

UserDetailsServiceImpl.java

```
@Service
public class UserDetailServiceImpl implements
                                        UserDetailsService {

    @Autowired
    private ClienteRepository repo;

    @Override
    public UserDetails loadUserByUsername(String username)
                            throws UsernameNotFoundException {
        Cliente cliente = repo.findByLogin(username);
        if (cliente == null) {
            throw new UsernameNotFoundException(username);
        }
        return new UserDetailsImpl(cliente.getId(),
                cliente.getLogin(), cliente.getSenha(),
                cliente.getPerfis());
    }
}
```

Definindo a chave secreta e o tempo de expiração do JWT

- O JWT exige uma chave secreta e o tempo de expiração de cada token
- Vamos definir essas informações no arquivo application.properties

```
jwt.secret=dfhdsf2483@@@#12dk
jwt.expiration=300000
```

Chave secreta

Tempo de expiração do token em milissegundos (5 minutos)

Criando a classe para gerar o token JWT

- Vamos implementar a classe JWTUtil com um método para gerar o token JWT

JWTUtil.java

```
@Component
public class JWTUtil {

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

    @Value("${jwt.expiration}")
    private Long expiration;

    public String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setExpiration(new Date(
                System.currentTimeMillis() + expiration))
            .signWith(SignatureAlgorithm.HS512,
                      secret.getBytes())
            .compact();
    }
}
```

Inicializa o atributo com o valor da propriedade no arquivo application.properties

CredenciaisDTO

- Vamos usar um DTO para transferir apenas o login e senha de um usuário para o *end point* de autenticação

CredenciaisDTO.java

```
@Getter
@Setter
@NoArgsConstructor
public class CredenciaisDTO implements Serializable {
    private static final long serialVersionUID = 1L;

    private String login;
    private String senha;
}
```

Criando um filtro de autenticação

- Herda da classe `UsernamePasswordAuthenticationFilter`
- Intercepta e processa requisições para /login
- Métodos
  - `attemptAuthentication`: executa a autenticação, caso bem sucedida, devolve um objeto `Authentication`
  - `successfulAuthentication`: põe o token JWT no cabeçalho da resposta
  - `unsuccessfulAuthentication`: devolve o status 401 (Unauthorized)

JWTAuthenticationFilter.java (1/2)

```
public class JWTAuthenticationFilter extends
                                    UsernamePasswordAuthenticationFilter {
   private AuthenticationManager authenticationManager;
   private JWTUtil jwtUtil;

   public JWTAuthenticationFilter(AuthenticationManager
                     authenticationManager, JWTUtil jwtUtil) {
      this.authenticationManager = authenticationManager;
      this.jwtUtil = jwtUtil;
   }

   @Override
   public Authentication attemptAuthentication(HttpServletRequest
 request, HttpServletResponse response) throws AuthenticationException {
      try {
         CredenciaisDTO creds = new ObjectMapper()
            .readValue(request.getInputStream(), CredenciaisDTO.class);
         UsernamePasswordAuthenticationToken authToken = new
            UsernamePasswordAuthenticationToken(creds.getLogin(),
                                    creds.getSenha(), new ArrayList<>());
         Authentication auth =
                authenticationManager.authenticate(authToken);
         return auth;
      } catch (IOException e) { throw new RuntimeException(e); }
   }
```

JWTAuthenticationFilter.java
(2/2)

```
@Override
protected void successfulAuthentication(HttpServletRequest request,
      HttpServletResponse response, FilterChain chain,
      Authentication authResult) throws IOException, ServletException {
   String username = ((UserDetailsImpl)  authResult.getPrincipal())
                                                .getUsername();
   String token = jwtUtil.generateToken(username);
   response.addHeader("Authentication", "Bearer " + token);
   response.addHeader("access-control-expose-headers","Authorization");
}

@Override
protected void unsuccessfulAuthentication(HttpServletRequest request,
         HttpServletResponse response, AuthenticationException failed)
         throws java.io.IOException, javax.servlet.ServletException {
   response.setStatus(401);
   response.setContentType("application/json");
   response.getWriter().append(jsonError());
}

private String jsonError() {
     return "{\"timestamp\": " + new Date().getTime() + ", "
            + "\"status\": 401, " + "\"error\": \"Não autorizado\", "
            + "\"message\": \"Email ou senha inválidos\", "
            + "\"path\": \"/login\"}";
}}
```

Registrando e configurando o filtro de autenticação (1/2)

- SecurityConfig.java

```
@Autowired
private JWTUtil jwtUtil;

@Autowired
private UserDetailsService userDetailsService;

...

@Override
protected void configure(HttpSecurity http)
                                        throws Exception {
    ...
    http.addFilter(new
      JWTAuthenticationFilter(authenticationManager(),
                                        jwtUtil));
}
```

Registra o filtro

Registrando e configurando o filtro de autenticação (2/2)

- SecurityConfig.java

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

Configura o AuthenticationManager para usar a implementação do UserDetailsService

Devolvendo mais informações sobre o usuário autenticado

- Além de devolver o token JWT no cabeçalho da resposta, o filtro de autenticação pode devolver essa e outras informações sobre o usuário autenticado
- Vamos mostrar como devolver um objeto JSON no corpo da resposta da autenticação

Classe JWTAuthenticationFilter.java

```
public class JWTAuthenticationFilter extends
                              UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;
    private JWTUtil jwtUtil;
    private ClienteRepository cliRepo;


    public JWTAuthenticationFilter(AuthenticationManager
            authenticationManager, JWTUtil jwtUtil,
            ClienteRepository cliRepo) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
        this.cliRepo = cliRepo;
    }
```

Não é possível usar o `@Autowired` em um filtro

Classe JWTAuthenticationFilter.java

```
public class JWTAuthenticationFilter extends
                              UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;
    private JWTUtil jwtUtil;
    private ClienteRepository cliRepo;


    public JWTAuthenticationFilter(AuthenticationManager
            authenticationManager, JWTUtil jwtUtil,
            ClienteRepository cliRepo) {
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;
        this.cliRepo = cliRepo;
    }
```

O repositório deve ser passado como parâmetro para o construtor para inicializar o atributo

Classe SecurityConfig.java

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    ...
    @Autowired
    private ClienteRepository cliRepo;
    ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {
       ...
       http.addFilter(new JWTAuthenticationFilter(
                   AuthenticationManager(), jwtUtil, cliRepo));
       ...
    }
    ...
}
```

Classe JWTAuthenticationFilter.java

```
@Override
protected void successfulAuthentication(HttpServletRequest request,
  HttpServletResponse response, FilterChain chain,
  Authentication authResult) throws IOException, ServletException {
    String username = ((UserDetailsImpl)
                        authResult.getPrincipal()).getUsername();
    String token = jwtUtil.generateToken(username);
    response.addHeader("Authentication", "Bearer " + token);
    response.addHeader("access-control-expose-headers",
                       "Authorization");
    Cliente cli = cliRepo.findByLogin(username);
    response.getWriter().append(jsonAuth(token, cli));
}

private String jsonAuth(String token, Cliente cliente) {
    return "{\"token\": \"" + token + "\"" + ", " +
            "\"username\": \"" + cliente.getNome() + "\", " +
            "\"profile\": " + cliente.getPerfis().stream()
                                .map(x -> "\"" + x + "\"")
                                .collect(Collectors.toList()) + "}";
}
```

Implementando a autorização

- Criar um filtro que herda da classe `BasicAuthenticationFilter`
- Responsável por processar qualquer requisição que possui um cabeçalho HTTP de `Authorization` com um esquema de autenticação básico e um *username* codificado em Base64: password token

JWTAuthorizationFilter.java (1/2)

```
public class JWTAuthorizationFilter extends
                                    BasicAuthenticationFilter {

   private JWTUtil jwtUtil;
   private UserDetailsService userDetailsService;

   public JWTAuthorizationFilter(AuthenticationManager
             authenticationManager, JWTUtil jwtUtil,
             UserDetailsService userDetailsService) {
       super(authenticationManager);
       this.jwtUtil = jwtUtil;
       this.userDetailsService = userDetailsService;
   }
```

JWTAuthorizationFilter.java
(2/2)

```
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                HttpServletResponse response, FilterChain chain)
                                throws IOException, ServletException {
       String header = request.getHeader("Authorization");
       if (header != null && header.startsWith("Bearer ")) {
          UsernamePasswordAuthenticationToken auth =
                               getAuthentication(header.substring(7));
          if (auth != null)
             SecurityContextHolder.getContext().setAuthentication(auth);
       }
       chain.doFilter(request, response);
    }

    private UsernamePasswordAuthenticationToken getAuthentication(
                                                        String token) {
       if (jwtUtil.tokenValido(token)) {
          String username = jwtUtil.getUsername(token);
          UserDetails user =
                     userDetailsService.loadUserByUsername(username);
          return new UsernamePasswordAuthenticationToken(user,
                                       null,user.getAuthorities());
       }
       return null;
    }
}
```

JWTUtil.java (1/2)

Verifica se o token JWT é válido

```
public boolean tokenValido(String token) {
    Claims claims = getClaims(token);
    if (claims != null) {
        String username = claims.getSubject();
        Date expirationDate = claims.getExpiration();
        Date now = new Date(System.currentTimeMillis());
        if (username != null && expirationDate != null
                            && now.before(expirationDate)) {
            return true;
        }
    }
    return false;
}
```

JWTUtil.java (2/2)

```
public String getUsername(String token) {
    Claims claims = getClaims(token);
    if (claims != null) {
        return claims.getSubject();
    }
    return null;
}

private Claims getClaims(String token) {
    try {
        return Jwts.parser().setSigningKey(secret.getBytes())
                   .parseClaimsJws(token).getBody();
    }
    catch (Exception e) {
        return null;
    }
}
```

`Claims` são atributos do usuário, extraídos da seção payload do token JWT

Registrando o filtro de autorização

- SecurityConfig.java

```
protected void configure(HttpSecurity http) throws
                                              Exception {
   http.cors().and().csrf().disable();
   http.authorizeRequests()
    .antMatchers(HttpMethod.GET, PUBLIC_MATCHERS)
    .permitAll()
    .antMatchers(HttpMethod.POST, PUBLIC_MATCHERS_POST)
    .permitAll().anyRequest().authenticated();
   http.addFilter(new JWTAuthenticationFilter(
                        authenticationManager(),jwtUtil));
   http.addFilter(new JWTAuthorizationFilter(
                     authenticationManager(), jwtUtil,
                     userDetailsService));
   http.sessionManagement().sessionCreationPolicy(
                   SessionCreationPolicy.STATELESS);
}
```

Autorizando *end points* para perfis específicos

- Adicionar a seguinte anotação no arquivo SecurityConfig.java
  - `@EnableGlobalMethodSecurity(prePostEnabled = true)`
- Anotar os métodos de recursos (*end points*) que você deseja liberar apenas para quem tem perfil de ADMIN com a anotação
  - `@PreAuthorize("hasAnyRole('ADMIN')")`

SecurityConfig.java

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends
                        WebSecurityConfigurerAdapter {
   ...
}
```

Todos os *end points* com acesso só para ADMIN

```
@Override
@DeleteMapping("/{id}")
@PreAuthorize("hasAnyRole('ADMIN')")
public ResponseEntity<?> delete(@PathVariable("id")
                                    Long id) {
   if (movService.delete(id)) {
      return ResponseEntity.ok().build();
   }
   return ResponseEntity
           .status(HttpStatus.NOT_FOUND).build();
}
```

Implementando restrições de conteúdo

- Pessoa física e pessoa jurídica só podem recuperar eles mesmos
- Os métodos `getById()` das classes `PessoaFisicaResources` e `PessoaJuridicaResources` só podem ser executados pelo próprio cliente ou por usuário com perfil de `ADMIN`

Recuperando o usuário autenticado

- Vamos implementar uma classe com um método estático para recuperar o usuário autenticado

```
@Service
public class ClienteService {

   public static UserDetailsImpl authenticated() {
      Authentication auth = SecurityContextHolder
                           .getContext().getAuthentication();
      if (auth != null) {
         return (UserDetailsImpl) auth.getPrincipal();
      }
      return null;
   }
}
```

Verificando o perfil do usuário

- Adicionar um método em `UserDetailsImpl` que verifica se o usuário tem um determinado perfil

```
public boolean hasRole(TipoPerfil perfil) {
    return getAuthorities()
      .contains(new SimpleGrantedAuthority(
                    perfil.getDescricao()));
}
```

Verificando se o usuário está autorizado

- Vamos implementar um método na classe **JWTUtil** que verifica se o usuário tem autorização para realizar a operação

```
public boolean authorized(Long id) {
    UserDetailsImpl user =
                        ClienteService.authenticated();
    if (user == null || (!user.hasRole(TipoPerfil.ADMIN)
         && !id.equals(user.getId()))) {
        return false;
    }
    return true;
}
```

Criando um exceção personalizada para erros de autorização

- Criar uma exceção que será lançada quando um usuário tentar acessar um recurso que ele não está autorizado

```
public class AuthorizationException extends
                                          RuntimeException {
   private static final long serialVersionUID = 1L;

   public AuthorizationException(String message) {
      super(message);
   }

   public AuthorizationException(String message,
                                 Throwable cause) {
      super(message, cause);
   }
}
```

Verificando se o usuário está autorizado a fazer a requisição

- Implementar a verificação nos métodos `findById()` na classe de serviços

```
@Autowired
private JWTUtil jwtUtil;
...
@Override
public PessoaFisicaDTO findById(Long id) throws
                                    AuthorizationException {
   if (!jwtUtil.authorized(id)) {
      throw new AuthorizationException(
                                       "Acesso negado!");
   }
   Optional<PessoaFisica> obj = repo.findById(id);
   if (obj.isPresent())
      return mapper.toDTO(obj.get());
   return null;
}
```

Tratando a exceção na classe de controle

- Tratar a exceção `AuthorizationException` no *end point* e retornar um erro

```
@GetMapping("/{id}")
public ResponseEntity<?> getById(@PathVariable("id") Long id)
{
    try {
        PessoaFisicaDTO obj = service.findById(id);
        if (obj != null) {
            return ResponseEntity.ok(_pf);
        }
        return ResponseEntity
                .status(HttpStatus.NOT_FOUND).build();
    } catch (AuthorizationException e) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                             .build();
    }
}
```

Obtendo um novo token

- Todo token expira
- Para obter um novo token sem a necessidade de se autenticar novamente, vamos implementar um *end point* para isso

AuthController.java

```
@RestController
@RequestMapping("/auth")
public class AuthController {
    @Autowired
    private JWTUtil jwtUtil;

    @PostMapping(value = "/refresh_token")
    public ResponseEntity<Void> refreshToken(
                                HttpServletResponse response) {
        UserDetailsImpl user = ClienteService.authenticated();
        if (user != null) {
            String token = jwtUtil
                            .generateToken(user.getUsername());
            response.addHeader("Authorization",
                             "Bearer " + token);
            return ResponseEntity.ok().build();
        }
        return ResponseEntity.status(HttpStatus.FORBIDDEN)
                             .build();
    }
}
```

Referências

- Spring Security. Disponível em: https://spring.io/projects/spring-security
- JSON Web Tokens (JWT). Disponível em: https://jwt.io/