AULA 20 - CONTROLE DE ACESSO

Disciplina de Backend - Professor Ramon Venson - SATC 2024

Controle de Acesso

O controle de acesso é um aspecto importante da segurança de aplicações web e tem como objetivo atingir garantias básicas a respeito dos dados trafegados por um sistema, como garantia de confidencialidade e integridade.

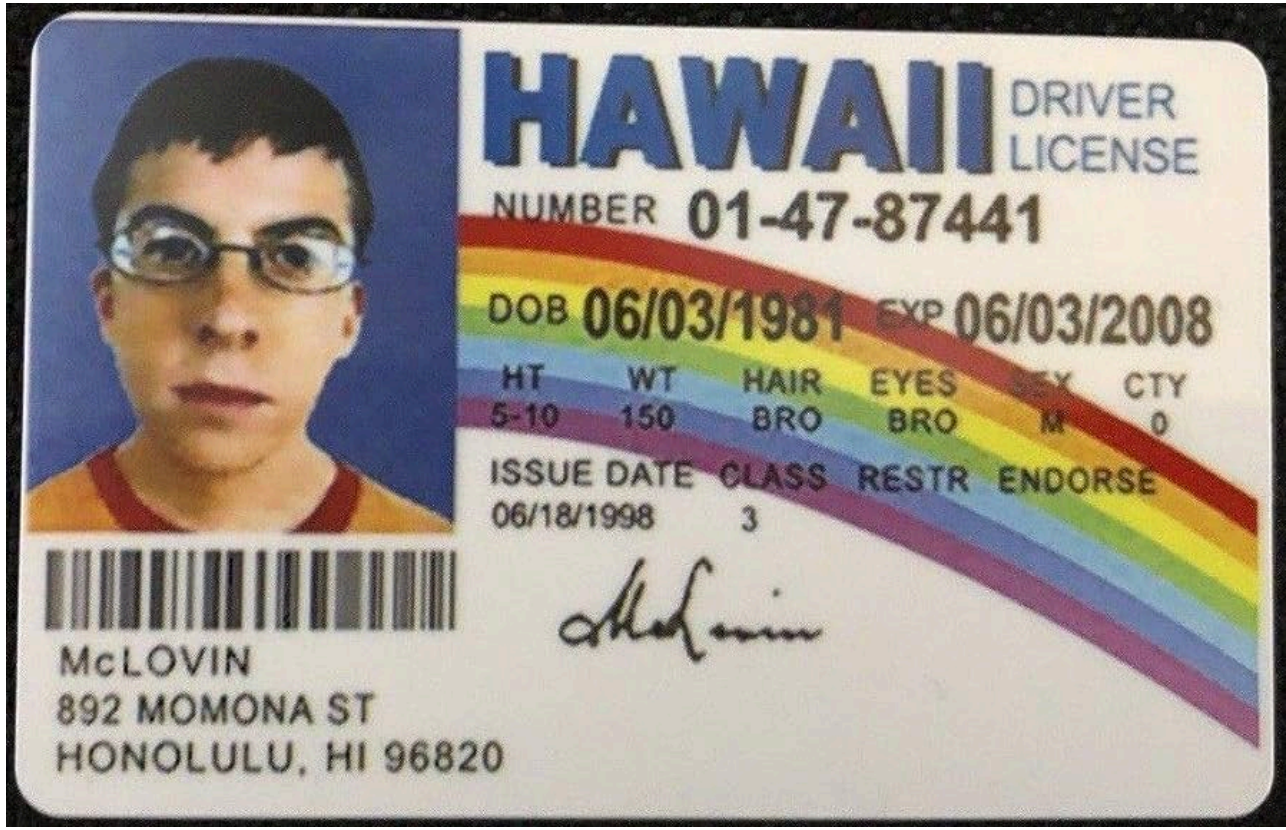
Um processo de controle de acesso é geralmente dividido em três diferentes fases:

- `Identificação`: quem é o requisitante?
- `Autenticação`: ele é conhecido?
- `Autorização`: o que ele pode fazer?

Identificação

O processo de identificação também pode ser entendido como parte da autenticação e tem como objetivo fornecer um formato para que o requisitante identifique-se.

Um usuário pode identificar-se usando nome de usuário, id, email ou dados biométricos.



Autenticação

O processo de autenticação tem como objetivo definir formas de provar a veracidade de uma identidade.

Esses formatos incluem senhas, tokens, certificados, dispositivos físicos ou dados biométricos que confirmem que a requisição veio realmente do usuário identificado.

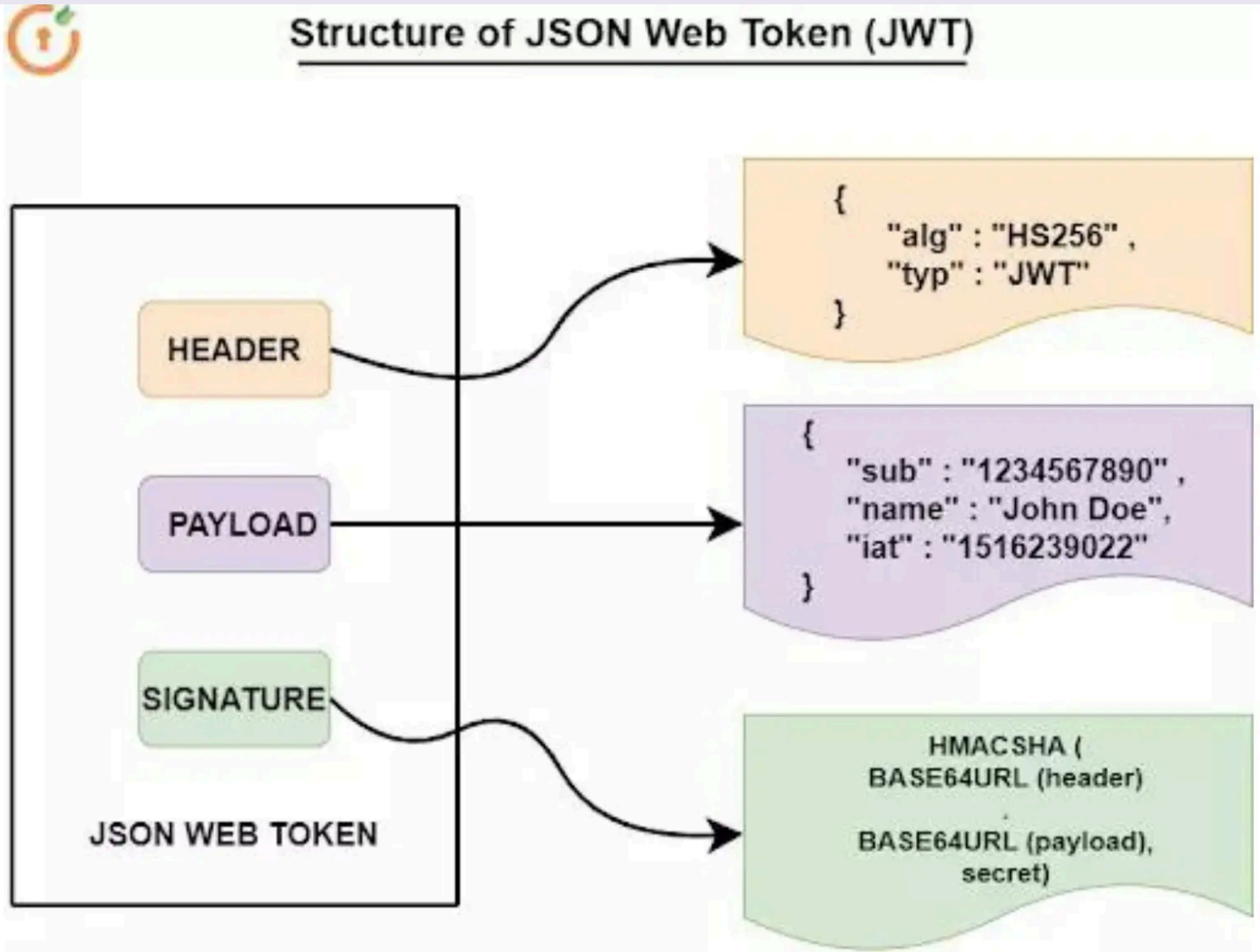
Autorização

Mesmo autenticado, um usuário/cliente pode não ter permissões suficientes para executar uma determinada ação no sistema. O processo de autorização verifica as permissões e dá a palavra final em termos de acesso a um recurso.

JWT

Um dos métodos mais utilizados para a implementação de aplicações web é o *JSON Web Tokens*. Essa tecnologia permite a criação de tokens assinados pela aplicação que podem ser validados para identificar e autenticar um usuário/cliente.

Um token JWT é um *hash* (*string*) que carrega informações como algoritmo usado no hash, payload (conteúdo) e a assinatura da aplicação.


Structure of JSON Web Token (JWT)
HEADER
PAYLOAD
SIGNATURE
JSON WEB TOKEN
{
"alg" : "HS256" ,
"typ" : "JWT"
}
{
"sub" : "1234567890" ,
"name" : "John Doe",
"iat" : "1516239022"
}
HMACSHA (
BASE64URL (header)
BASE64URL (payload),
secret)

Um token é gerado pelo servidor ao realizar a autenticação de um usuário.

O token deve ser enviado de volta pelo cliente no cabeçalho de uma requisição, substituindo o envio de usuário e senha.

Ao verificar um token, uma aplicação pode confirmar que um usuário está autenticado (logado), não sendo necessário validar novamente seus dados de usuário e senha.

Essa estrutura do JWT dispensa que um usuário/cliente precise enviar informações sigilosas, como sua senha, todas as vezes que precisar realizar uma requisição. O token deve ser sempre armazenado pelo cliente.

Outra vantagem do JWT é garantir que diferentes aplicações possam trabalhar com o mesmo token, já que pra isso só é necessário que todas compartilhem um mesmo `secret` (palavra-chave) usado na geração dos tokens.

Um token JWT não deve conter dados sigilosos, pois seu conteúdo não é *encriptado* e sim *codificado* e *assinado*. Para efeitos técnicos, um token é assinado com uma chave privada para que

Um token ainda deve, preferencialmente, conter informações de expiração (data em que não será mais válido), com essa data sendo a menor possível.

Logouts e re-logins são geralmente gerenciados apenas localmente, pelo cliente. Para o servidor, um *token* é valido até que expire. **Um token geralmente não é salvo no lado do servidor**.

Por fim, é importante lembrar que para boa parte dos casos, um token JWT não exclui a necessidade de verificar um usuário no banco de dados a cada requisição.

Adicionar permissões e papéis (*roles*) no conteúdo de um *token* para diminuir chamadas à base de dados não é recomendado a menos que seja um requisito.

Também é possível criar sistemas de *blocklist* para rejeitar tokens já gerados. Não é possível revogar manualmente um token JWT a menos que se altere a geração.

Resumo de práticas para o JWT

- JWT é um método para criação de tokens para autenticação;
- O servidor gera tokens que devem ser armazenados pelo cliente para as requisições;
- Um token pode conter um conteúdo personalizado (*payload*);
- Um token deve possuir tempo de expiração e não pode ser revogado manualmente;
- Não incluir informações sigilosas no token;
- Um token não necessariamente vai reduzir o número de chamadas ao banco de dados;

Spring Security

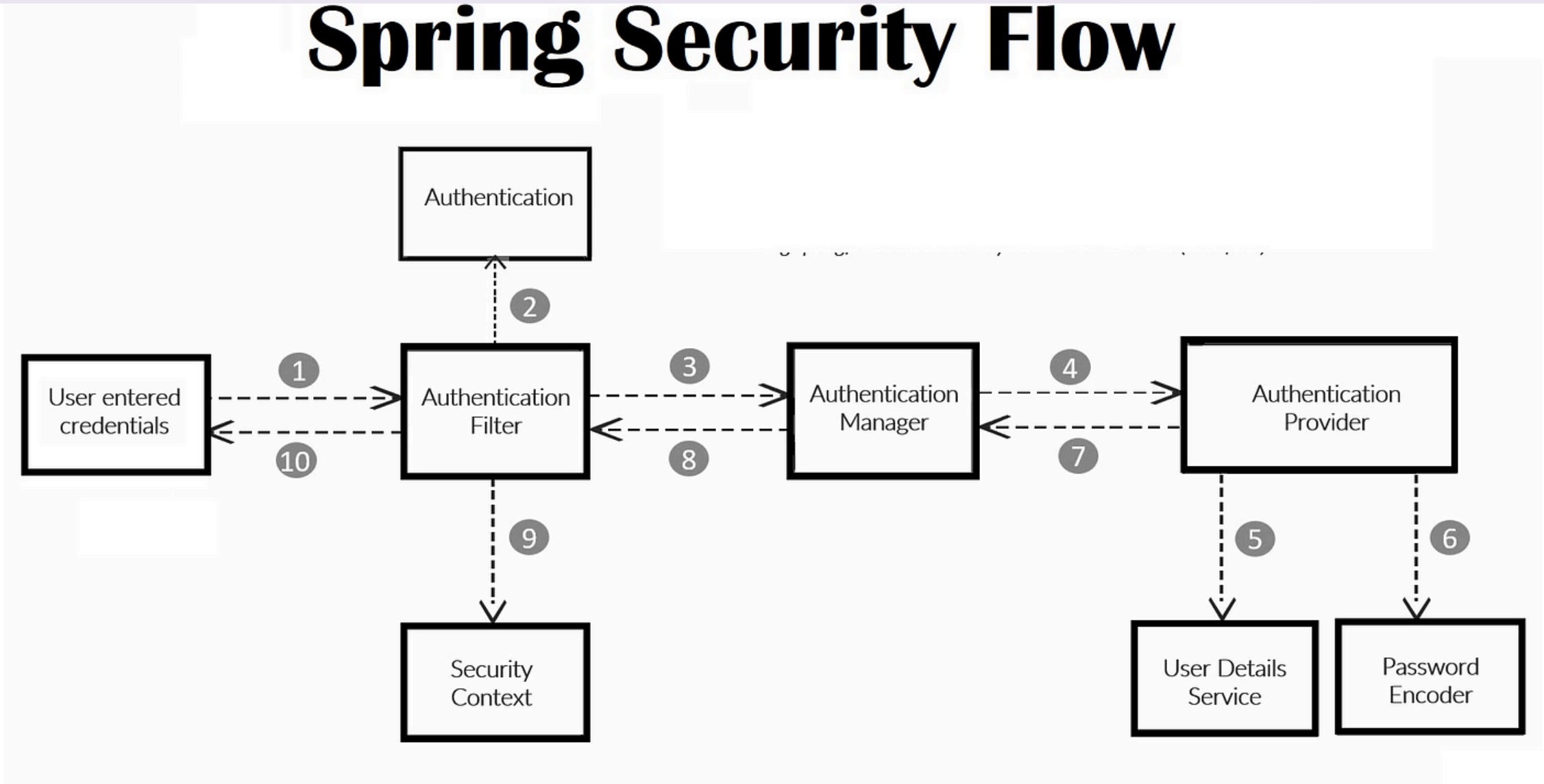
O Spring Security é um framework de controle de acesso e autenticação desenvolvido como um módulo para o Spring. Ele permite que o controle de acesso seja implementado rapidamente em uma aplicação, com uma grande liberdade de definição da arquitetura e dos métodos de autenticação.

A implementação do Spring Security pode ser um pouco complexa dado seu fluxo de operação contar com diferentes formas de autorização/autenticação.

Porém as tarefas para adicionar uma autenticação simples com usuário e senha em uma nova aplicação passam por:

- Adicionar os pacotes Spring Security e o JWT às dependências (`pom.xml`);
- Criar um modelo de usuário (`User`) e seu respectivo repositório;
- Criar DTOs para o recebimento e envio de respostas ao cliente;
- Definir um serviço de geração/validação de tokens JWT;

- Definir configurações dos tokens no `application.properties`;
- Implementação do serviço de autenticação com métodos para login e registro;
- Implementação de um filtro para validar os tokens em cada requisição (SecurityFilter);
- Implementação das configurações de autorização (`SecurityConfig`);
- Implementação de um `AuthorizationService`;
- Adicionar os controladores com `/login` e `/senha`.


Spring Security Flow
Authentication
2
User entered credentials
1
10
Authentication Filter
3
8
Authentication Manager
4
7
Authentication Provider
9
5
6
Security Context
User Details Service
Password Encoder

Adicionando Dependências

Vamos adicionar duas dependências ao nosso arquivo `pom.xml`. O código abaixo deve ser adicionado dentro da tag `<dependencies>`:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
</dependency>
<dependency>
    <groupId>com.auth0</groupId>
    <artifactId>java-jwt</artifactId>
    <version>4.4.0</version>
</dependency>
```

A instalação pode ser feita pelo plugin da IDE ou via comando: `./mvnw install`

Adicionando o modelo `User`

Para realizarmos a autenticação via Usuário e Senha no Spring Security, vamos precisar de um modelo para armazenar os dados de cada usuário, incluindo `username`, `password` e `roles`.

Além disso, será essencial fazer com que essa classe implemente a classe `UserDetails` e seus respectivos métodos. O Spring utiliza os métodos implementados para realizar a autenticação e autorização.

```java
@Entity
@Table(name = "users")
@Data
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID id;
    @Column(unique = true)
    private String username;
    @Column(unique = true)
    private String email;
    private String password;
    private Set<String> roles = new HashSet<>();
}
```

Não esqueça de implementar os métodos herdados de `UserDetails`. Alguns desses métodos devem retornar os papéis (*roles*), usuário e senha.

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return roles.stream().map((role -> new SimpleGrantedAuthority(role))).toList();
}

@Override
public String getPassword() {
    return this.password;
}

@Override
public String getUsername() {
    return this.username;
}
```

Estes outros métodos devem retornar `true` caso não exista controle de acesso à conta, caso contrário o usuário não terá permissão para realizar o 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; }
```

Criando o repositório

Vamos criar um repositório para o modelo `User`, incluindo dois métodos para realizar a busca por `username` e `email`, que podem ser úteis na hora de registrar e encontrar o usuário a ser autenticado/autorizado.

```
public interface UserRepository extends JpaRepository<User, UUID> {
    Optional<User> findByUsername(String username);
    Optional<User> findByEmail(String email);
}
```

Criando os DTOs

Vamos criar 4 (quatro) records diferentes para transferir dados entre o cliente e a aplicação:

- `LoginRequestDto`: contendo as informações que o cliente envia para login;
- `LoginResponseDto`: contendo o *JSON Web Token* (JWT) de autenticação;
- `RegisterDto`: contendo as informações que o cliente envia para registro de um novo usuário;
- `UserDto`: contendo as informações de um usuário (sem o `password`);

`LoginRequestDto`

```
public record LoginRequestDto(String username, String password) {}
```

Exemplo de Dado

```
{
    "username": "admin",
    "password": "123456"
}
```

`LoginResponseDto`

```
public record LoginResponseDto(String token) {}
```

Exemplo de Dado

```
{
    "token": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.
    eyJpc3MiOiJydmVuc29uIiwic3ViIjoicnZlbnNvbiIsImlhdCI6MTcxNzQyNTMyOSwiZXhwIjoxNzE3NDM5NzI5fQ.
    gePF9Q28GzymFwLPfqMms5gbReT5EteXEygQ_WvXxQioj8yP6_t0XeTMwNHvDtbcPuDEwGE2Van6cGD97YKzVQ"
}
```

`RegisterDto`

```
public record RegisterDto(String username, String email, String password) {}
```

Exemplo de Dado

```
{
    "username": "admin",
    "email": "admin@localhost.com.br",
    "password": "123456"
}
```

`UserDto`

```
public record UserDto(String username, String email) {}
```

Exemplo de Dado

```
{
    "username": "admin",
    "email": "admin@localhost.com.br"
}
```

Gerando e Validando Tokens

O usuário que realiza autenticação sempre recebe um *token* JWT, que deve ser validado à cada requisição pela nossa aplicação. Criaremos um novo service chamado `TokenService` com dois métodos para gerar e validar todos os *tokens*.

Aqui vamos utilizar também algumas variáveis de ambiente, como o `secret` (chave secreta usada para gerar o token), o `issuer` (identificador de quem está gerando) e `expirationMinutes` (define por quanto tempo o *token* será válido).

```
@Service
public class TokenService {
    @Value("${api.security.token.secret}")
    private String secret;
    @Value("${api.security.token.issuer}")
    private String issuer;
    @Value("${api.security.token.expiration-minutes}")
    private long expirationMinutes;

    public String generateToken(String username) {}
    public String validateToken(String token) {}
}
```

Lembre-se de adicionar essas variáveis ao seu `application.properties` ou adicionar como variáveis de ambiente na execução do sistema.

```
api.security.token.secret=${JWT_SECRET:default_token_secret}
api.security.token.issuer=${JWT_ISSUER:rvenson}
api.security.token.expiration-minutes=${JWT_EXPIRES_MINUTES:240}
logging.level.org.springframework.security=DEBUG
```

O nível de debug do Spring Security também pode ser alterado usando o parâmetro `DEBUG`.

Na classe `TokenService`, vamos começar com o método para geração do token:

```
public String generateToken(String username) {
    Algorithm algorithm = Algorithm.HMAC512(this.secret);
    Instant issuedAt = Instant.now().truncatedTo(ChronoUnit.SECONDS);
    Instant expiration = issuedAt.plus(expirationMinutes, ChronoUnit.MINUTES);
    try {
        String token = JWT.create()
                .withIssuer(this.issuer)
                .withSubject(user.getUsername())
                .withIssuedAt(issuedAt)
                .withExpiresAt(expiration)
                .sign(algorithm);
        return token;
    } catch (JWTCreationException exception) {
        throw new RuntimeException("Error while generating the JWT token");
    }
}
```

O método recebe o nome de usuário e sempre retorna um token válido a ser utilizado por ele. O *payload* (*subject*) definido em `.withSubject` é o nome de usuário, para que possamos recuperar no futuro.

Em caso de erro com o processo, usamos o `try-catch` para gerar um erro customizado. Um erro é causado quando não é possível gerar um token válido com essa configuração, como por exemplo (mas não limita-se a):

- Algoritmo é inválido
- Payload (subject) é nulo
- Tamanho do token inválido

No método de validação de um token, vamos recuperar o nome de usuário a partir do token informado:

```
public String validateToken(String token) {
        try {
            Algorithm algorithm = Algorithm.HMAC512(this.secret);
            String username = JWT
                    .require(algorithm)
                    .withIssuer(this.issuer)
                    .build()
                    .verify(token)
                    .getSubject();
            return username;
        } catch (JWTDecodeException exception) {
            throw new RuntimeException("Error while decoding the JWT token");
        }
    }
```

Alerta

Repare que para recuperar o nome de usuário a partir do token, precisamos do `issuer` e do `secret`. Ainda que o `secret` seja sigiloso e necessário para que o *token* seja validado pela aplicação, **as informações contidas no token podem ser facilmente identificadas**. Por isso, não insira informações sigilosas como senha ou papéis do usuário no conteúdo (subject) de um *token* JWT.

Você pode ver o conteúdo de um JWT em sites como https://jwt.io/

Implementando o Serviço de Autenticação

Agora com a geração/validação de tokens implementada, vamos implementar um serviço que será o responsável por duas ações bem específicas na nossa aplicação: o login e o registro de novos usuários.

Nesse serviço, além do `TokenService`, vamos precisar de acesso ao `UserRepository` e do `AuthenticationManager`, uma classe do próprio Spring Security que vamos configurar depois.

```
@Service
public class AuthenticationService {
    @Autowired
    AuthenticationManager authenticationManager;
    @Autowired
    UserRepository userRepository;
    @Autowired
    TokenService tokenService;

    @Transactional(readOnly = true)
    public LoginResponseDto login(LoginRequestDto authenticationDto)
    throws AuthenticationException {
    }

    @Transactional
    public UserDto register(RegisterDto registerDto) {
    }
}
```

Login

Na implementação do login, vamos repassar o `LoginRequestDto` ao `AuthenticationManager` para verificar se o usuário está correto e autenticá-lo. Geramos um novo token do usuário que foi autenticado usando `getPrincipal()`:

```
@Transactional(readOnly = true)
    public LoginResponseDto login(LoginRequestDto loginRequestDto)
    throws AuthenticationException {
    Authentication usernamePassword = new UsernamePasswordAuthenticationToken(
            loginRequestDto.username(), loginRequestDto.password()
    );
    Authentication auth = authenticationManager.authenticate(usernamePassword);
    User user = (User) auth.getPrincipal();
    String token = tokenService.generateToken(user.getUsername());
    return new LoginResponseDto(token);
}
```

Registro

No processo de registro, usamos as informações do `RegisterDto` simplesmente para criar um novo usuário. Não é necessário autenticar ou gerar token nessa etapa.

```
@Transactional
public UserDto register(RegisterDto registerDto) {
    User user = new User();
    user.setUsername(registerDto.username());
    user.setEmail(registerDto.email());
    user.setPassword(new BCryptPasswordEncoder().encode(registerDto.password()));
    User savedUser = userRepository.save(user);
    return new UserDto(savedUser.getUsername(), savedUser.getEmail());
}
```

Repare que o novo usuário será criado com o atributo `roles` vazio.

SecurityFilter

Para realizar a autorização de uma nova requisição, o Spring Security deve interceptar e verificar as permissões necessárias para continuar a execução da requisição.

Por exemplo, algumas rotas da nossa aplicação devem ser abertas à requisições "anônimas" (sem *token*), enquanto outras podem requerer usuários autenticados ou até permissões específicas.

Dessa forma, criaremos um filtro que será executado antes que as requisições cheguem ao seu controlador (`@RestController`).

```java
@Component
public class SecurityFilter extends OncePerRequestFilter {
    @Autowired
    TokenService tokenService;
    @Autowired
    UserRepository userRepository;

    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain)
        throws ServletException, IOException {
            // Esse filtro é executado em cada requisição pois extende
            // a class OncePerRequestFilter (Uma vez por requisicao)
    }
}
```

```
@Override
    protected void doFilterInternal(
        HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        String token = this.recoverToken(request);
        if (token != null) {
            String subject = tokenService.validateToken(token);
            User user = userRepository.findByUsername(subject).get();
            UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(
                    user,
                    user.getAuthorities(),
                    user.getAuthorities()
            );
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(request, response);
    }
```

O método `recoverToken` pode ser implementado como um método privado na mesma classe:

```
private String recoverToken(HttpServletRequest request) {
    String authHeader = request.getHeader("Authorization");
    if (authHeader == null) return null;
    return authHeader.replace("Bearer ", "");
}
```

Esse método serve apenas para encapsular a extração do token do cabeçalho de uma requisição, que por padrão vem no formato:

```
Bearer eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJydmVuc29u...
```

SecurityConfig

A classe `SecurityConfig` será utilizada para realizar configurações do filtro recém criado. Essas configurações incluem a definição do método de autenticação e das permissões necessárias para cada rota dentro da nossa aplicação:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    SecurityFilter securityFilter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity)
    throws Exception {
    }
}
```

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    return httpSecurity
            .csrf(csrf -> csrf.disable())
            .sessionManagement(session -> session.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS))
            .addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
            .authorizeHttpRequests(authorize -> authorize
                    .requestMatchers("/auth/*").permitAll()
                    .requestMatchers(HttpMethod.GET, "/").authenticated()
                    .requestMatchers(HttpMethod.POST, "/").hasAuthority("ROLE_MANAGER")
                    .anyRequest().hasAuthority("ROLE_ADMIN")
            )
            .build();
}
```

Nessa configuração, especificamos que todas as rotas de `/auth/` serão permitidas à usuários não logados. Todas as rotas GET serão permitidas apenas à usuários autenticados. Todas as rotas POST serão permitidas à usuários manager. Todas as demais rotas serão permitidas apenas a administradores.

Também vamos adicionar alguns Beans que de objetos que são automaticamente utilizados pelo Spring no processo de autorização:

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
throws Exception {
    return authenticationConfiguration.getAuthenticationManager();
}

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

AuthorizationService

Vamos criar também um *service* chamado `AuthorizationService` que vai retornar automaticamente um usuário pelo seu nome quando o Spring Security precisar realizar a validação de um *token*.

```
@Service
public class AuthorizationService implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userRepository.findByUsername(username).get();
    }
}
```

Implementando o Controller

Por fim, implementaremos um controlador chamado `AuthenticationController` para que os clientes possam finalmente realizar as ações de registro e login.

```
@RestController
@RequestMapping("/auth")
public class AuthenticationController {
    @Autowired
    AuthenticationService authenticationService;
}
```

Rota de Login

```
@PostMapping("/login")
@ResponseStatus(HttpStatus.OK)
public LoginResponseDto login(@RequestBody @Valid LoginRequestDto authenticationDto) {
    return authenticationService.login(authenticationDto);
}
```

Rota de Registro

```
@PostMapping("/register")
@ResponseStatus(HttpStatus.CREATED)
public UserDto register(@RequestBody @Valid RegisterDto registerDto) {
    return authenticationService.register(registerDto);
}
```

Funcionamento da Segurança

Para cadastrar um novo usuário, deve-se realizar uma requisição para `/auth/register`. Essa rota retorna apenas algumas informações do usuário recém cadastrado.

Em seguida, realize o login de um usuário cadastrado usando `/auth/login`. Essa rota retorna o token do usuário validado.

Para as próximas requisições, adicione ao cabeçalho (header) um cabeçalho chamado `Authorization`, e em seu conteúdo adicione o token recebido no processo de login junto do prefixo `Bearer `.

Ex.: `Authorization: Bearer eyJhbG...`

Adicione novas permissões ao usuário usando o atributo `roles`. Essas permissões podem estar no formato `ROLE_*` (ex.: `ROLE_ADMIN`). Como não foi implementado nenhuma rota para modificar as permissões, isso pode ser feito diretamente no banco de dados utilizado.

O que aprendemos hoje

- O que é autorização e autenticação
- O que é o JWT
- Como realizar o processo de autenticação e autorização usando Spring Security