

Desenvolvimento de APIs REST

10 - Autenticação

- Autenticação Basic HTTP
- CSRF e CORS
- Permissão de Acesso
- JWT

Autenticação e Autorização



Autenticação

Serve para verificar quem é o usuário. Podemos ter autenticação de várias formas:

- Baseada no nome de usuário e senha
- Autenticação sem senha
- Autenticação multifator
- Baseada em impressão digital ou íris, etc.

JWT é um padrão para autenticação.

Autorização

Determinar o que um usuário tem permissão para fazer em uma determinada aplicação

OAuth é um padrão aberto para autorização.

Basic Authentication



Autenticação HTTP Basic

É a forma básica de autenticar um usuário na API, para isto precisamos definir um usuário e senha no servidor. O Spring possui essa autenticação embutida automaticamente no Spring Security.

Spring Security

É uma biblioteca que fornece autenticação e autorização trabalhando com diversos protocolos. Precisamos adicionar a dependência do Spring Security para trabalhar com a autenticação.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Quando utilizar autenticação básica

Geralmente usada em testes de desenvolvimento ou também quando queremos integrar nossa aplicação com outras aplicações de forma rápida.

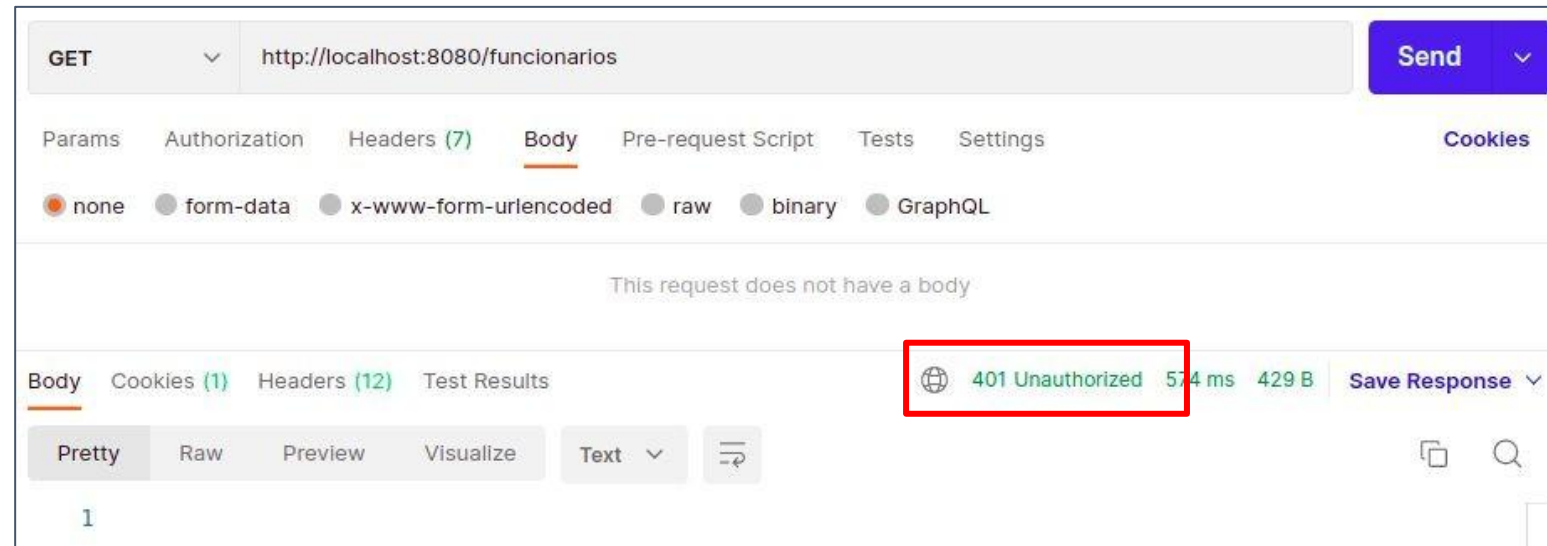
Basic Authentication



Vamos incluir a dependência **spring security** na aplicação service-dto. Ao executarmos a aplicação, no console será exibida um usuário e senha padrão definida pelo Spring Security.

```
2022-09-06 14:35:27.052 WARN 230181 --- [ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :  
Using generated security password: daf110f8-c06c-44f7-9c0b-0b0af9635c26  
This generated password is for development use only. Your security configuration must be updated before running the application.  
2022-09-06 14:35:27.149 INFO 230181 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Will secure the following URLs: /  
2022-09-06 14:35:27.245 INFO 230181 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 8080 (http) with context path ''
```

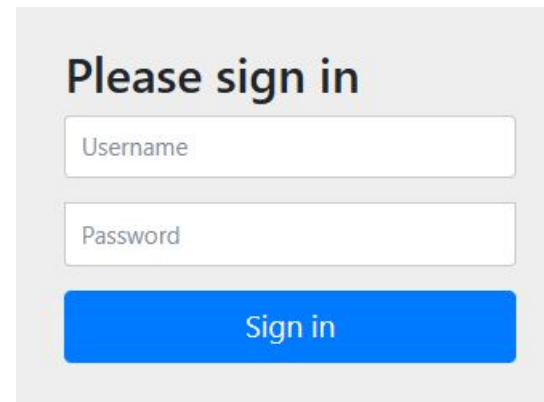
Quando testarmos no Postman, o código retornado será **“401 não autorizado.”**



Basic Authentication



Quando abrimos o browse e digitamos **http://localhost:8080** para tentarmos acessar um recurso o Spring Security exibe uma tela de login padrão

The image shows a standard Spring Security login page. It has a light gray background. At the top, the text "Please sign in" is displayed in a bold, dark font. Below this, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both fields are white with a thin gray border. Below the password field is a blue button with the text "Sign in" in white.

Digite **user:** e cole a senha gerada pelo console para acesso.

Obs: **user** é o usuário padrão definido pelo Spring

Basic Authentication



No **Postman** selecione a opção **headers** e no campo **Key** digite **Authorization**. No campo **Value** digite **Basic** dê um espaço e cole a senha gerada pelo console

Authorization: Basic daf110f8-44f7-9c0b-0b0af9635c26

The screenshot shows the Postman interface for a GET request to `http://localhost:8080/clientes`. The **Headers** tab is selected, showing a table with the following headers:

Key	Value	Description
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Authorization	Basic daf110f8-44f7-9c0b-0b0af9635c26 ..	

The **Body** tab is also visible, showing a JSON response in the **Pretty** view:

```
1 [
2   {
3     "id": 1,
4     "nome": "Joaquina",
5     "cpf": "943536",
6     "email": "joaquim@gmail.com",
7     "endereco": {
8       "logradouro": "Rua Fonseca Ramos",
9       "numero": "134",
10      "bairro": "Centro",
11      "cidade": "Juiz de Fora",
12      "estado": "RJ"
13    }
14  },
15 ]
```

The response status is **200 OK** with a response time of **1203 ms** and a size of **800 B**. The **Save Response** button is visible.

Basic Authentication Configuração de Segurança



Classe e Métodos de configuração

Vamos criar a classe de configuração **ConfigSeguranca**, nessa classe vamos incluir duas anotações **@Configuration** e **EnableWebSecurity**. A anotação **@EnableWebSecurity** serve para habilitar a segurança para os serviços web. Em versões anteriores do Spring Security havia a necessidade de herdamos de **WebSecurityConfigurerAdapter** a partir da **versão 6** essa classe não é mais disponibilizada.

Como é uma classe de configuração vamos precisar de um Bean para uso do Filter que vai interceptar uma requisição para checar se ela é autenticada ou não. Vamos retornar a classe **SecurityFilterChain** que é uma cadeia de filtros de segurança. Vamos armazenar o usuários em memória utilizando o método **userDetailsService** que permite as seguintes configurações: nome do usuário, senha e perfil.

Autenticação em Memória

Estamos dizendo que qualquer requisição autenticada e usando autenticação básica.

Devemos utilizar o usuário e senha informados para acesso a aplicação, desta forma, o spring security não irá gerar mais senha no console.

Abra o browser digite <http://localhost:8080> e faça o teste

```
@Configuration
@EnableWebSecurity
public class ConfigSeguranca {
```

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(requests -> requests.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults());
        return http.build();
    }
```

```
    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder().
            username("teste").
            password("123456").
            roles("RH").build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Importação

```
org.springframework.security.config.Customizer;
```

Basic Authentication Configuração de Segurança



Método `configure(HttpSecurity http)`

Permite a alteração de configurações relativas às requisições do protocolo HTTP, como autenticação, configuração stateless, e outras configurações.

Para este projeto serão utilizadas as seguintes configurações:

- Definição do HTTP Basic como método de autenticação
- Definição das rotas que deverão ser autenticadas ou não
- Configuração de gerenciamento de sessão

vamos modificar o método **filterChain**

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.httpBasic(Customizer.withDefaults()).authorizeHttpRequests(requests -> {
        requests.requestMatchers(HttpMethod.GET, "/funcionarios").permitAll();
        requests.requestMatchers(HttpMethod.GET,
            "/usuarios").hasRole("RH").anyRequest().authenticated();
    }).sessionManagement(session ->
        session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```

O recurso **/funcionarios** no método **GET** qualquer usuário pode acessar o recurso

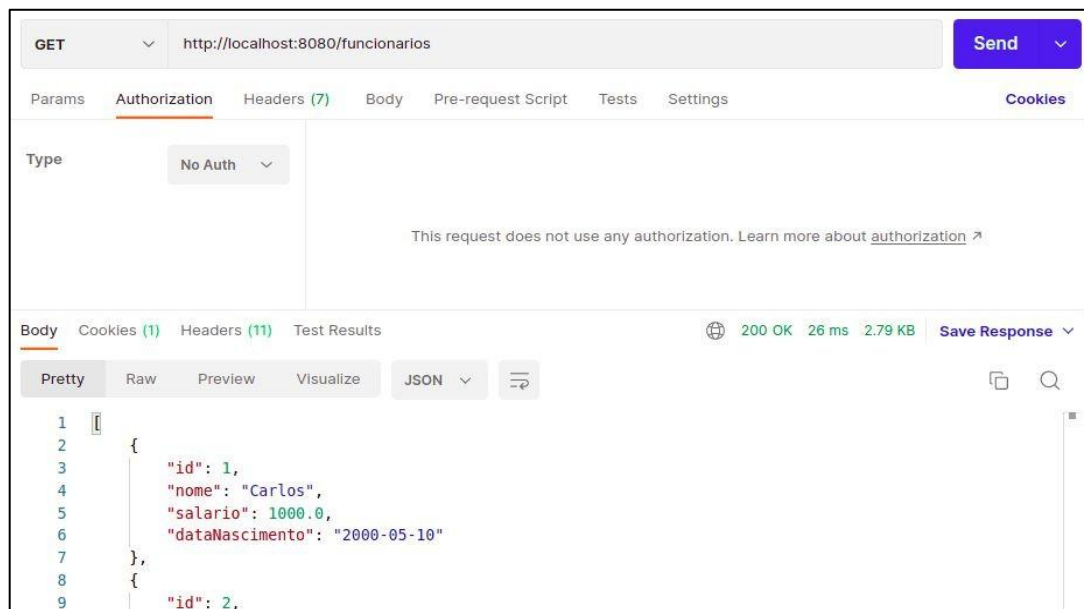
O recurso **/usuarios** no método **GET** somente usuários no perfil RH poderão acessar de forma autenticada.

Nosso servidor também não vai guardar a sessão.
(**SessionCreationPolicy.STATELESS**)

Basic Authentication Configuração de Segurança

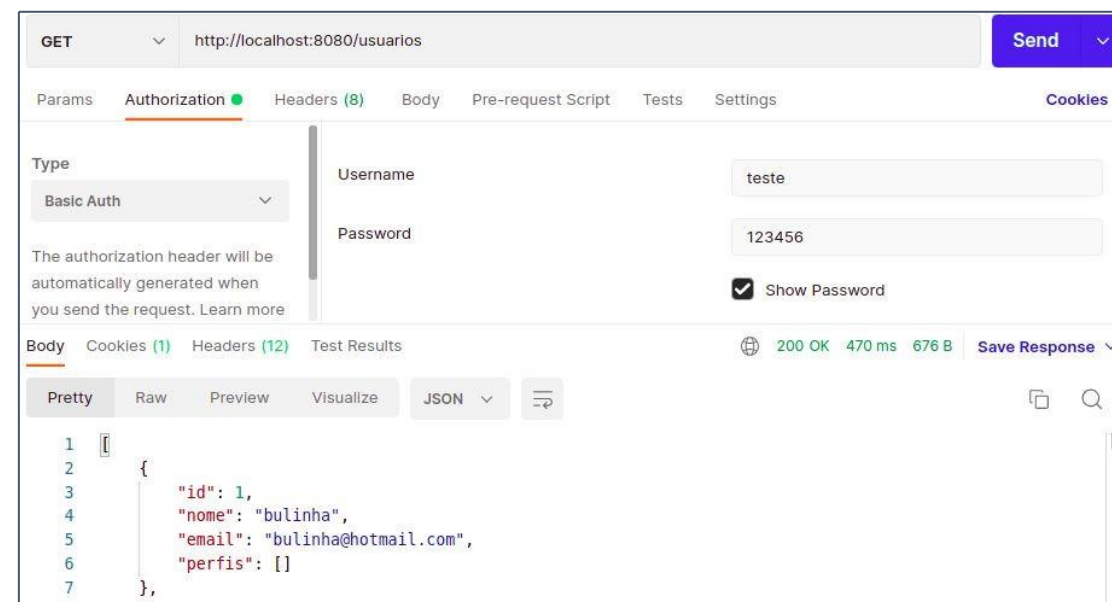
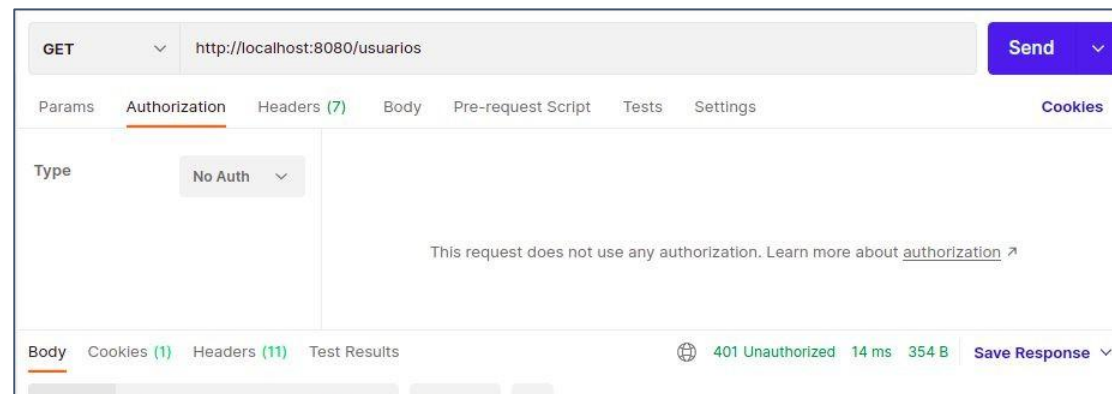


Vamos testar no Postman limpando os campos **key** e **value** da aba **Headers** e clicar na aba **Authorization**, selecione **No Auth**



Conseguimos acessar **/funcionarios** pois eles estão com **permiteAll()** e sem autenticação como uma rota pública na configuração, ou seja, qualquer usuário pode acessar, mesmo não estando autenticado.

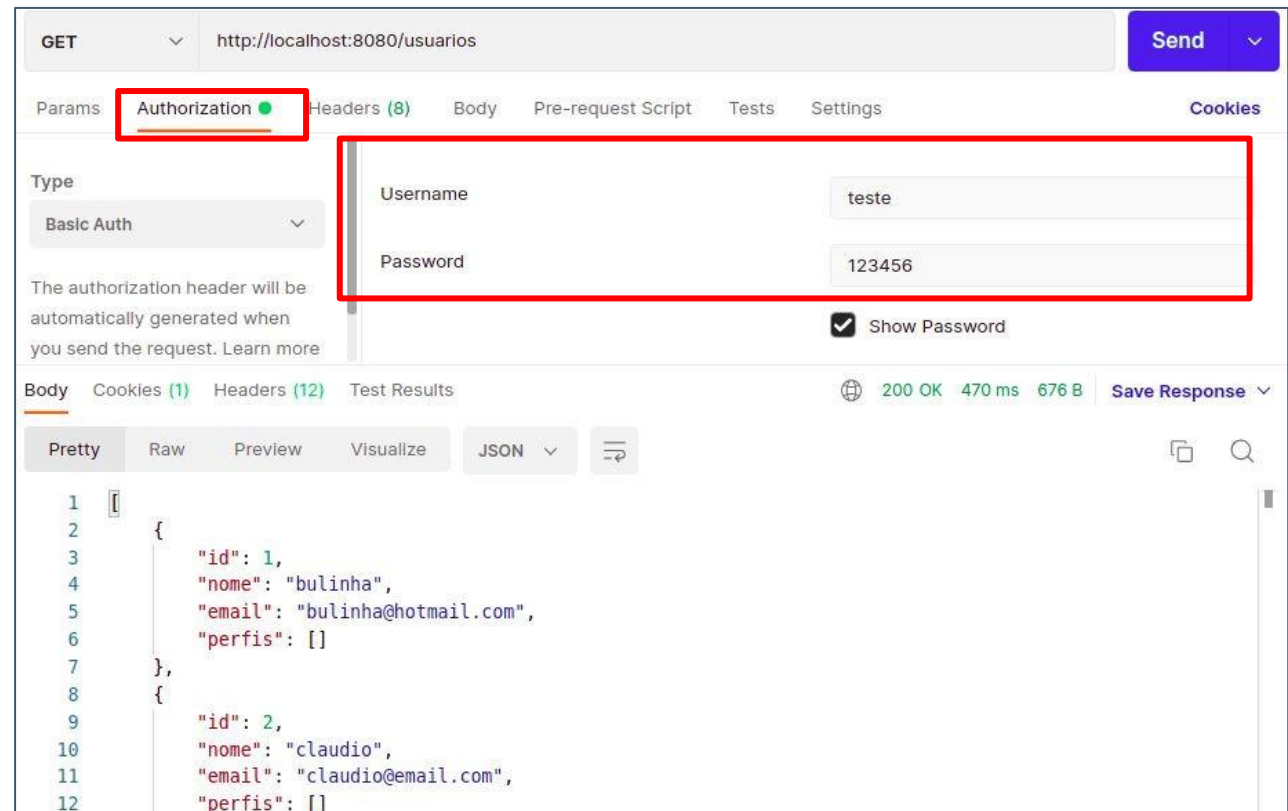
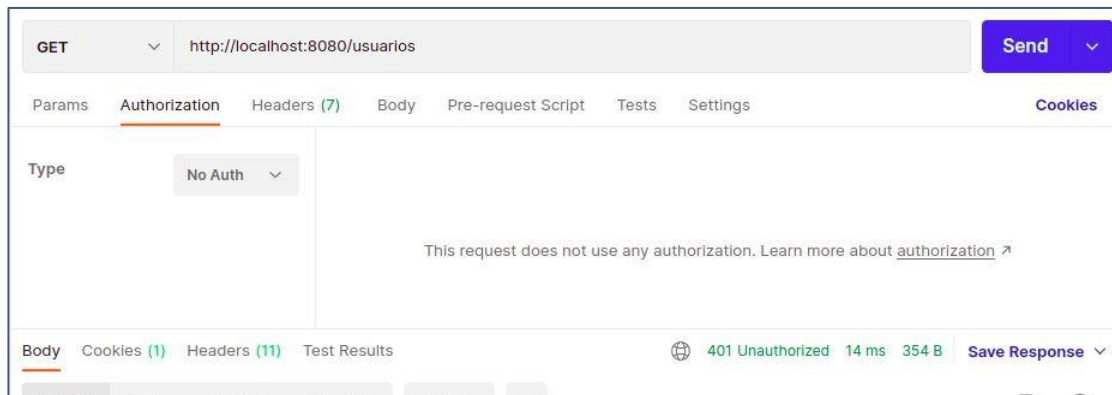
Ao tentarmos acessar **/usuarios** recebemos o código de retorno **401**, pois é necessário estar autenticado **anyRequest().authenticated()**



Configuração de Segurança



Vamos clicar na aba **Authorization**, selecionar **Basic Auth** inserir o usuário teste e a senha 123456 para conseguir listar os usuarios.



Recuperando o Usuario autenticado

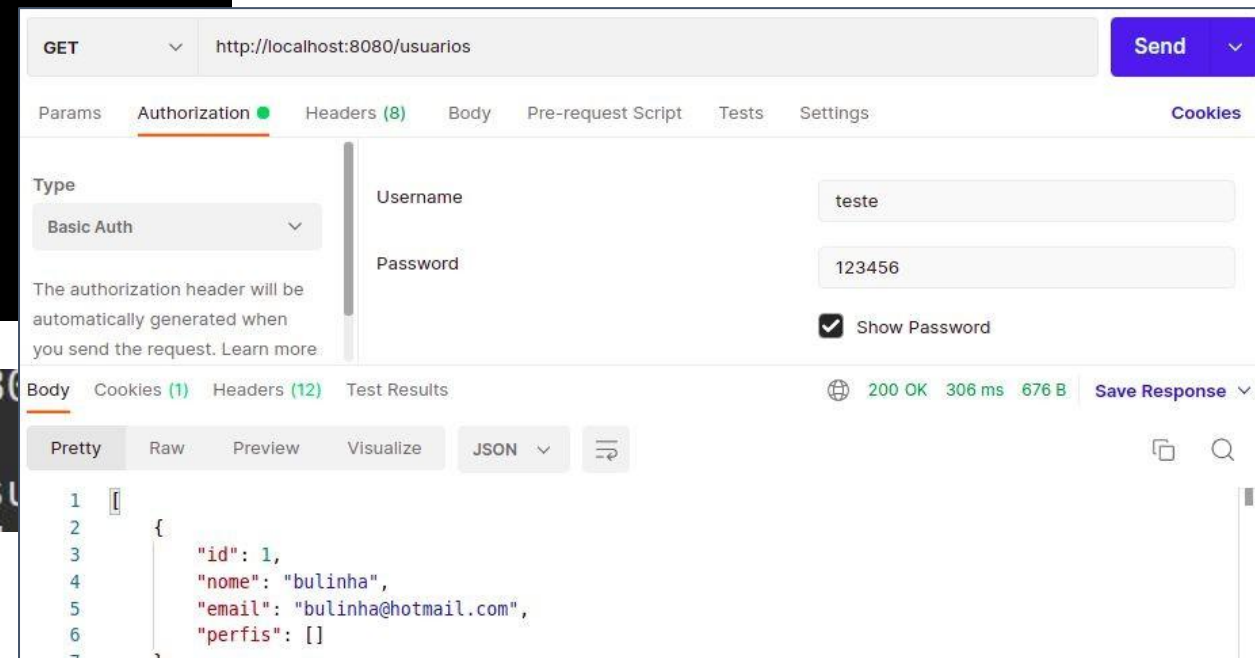


Usando a anotação **@AuthenticationPrincipal**, o Spring Boot pode passar como parâmetro o objeto **UserDetails** que tem informações sobre o usuário autenticado.

Não há necessidade de passar nenhuma informação além do header **Authorization**

```
@GetMapping
public ResponseEntity<List<UsuarioDTO>> listar(
    @AuthenticationPrincipal UserDetails details) {
    System.out.println("Login do usuario: " + details.getUsername());
    return ResponseEntity.ok(usuarioService.findAll());
}
```

```
2022-09-07 19:09:17.247 INFO 553169 --- [nio-8080]
Login do usuario: teste
Hibernate: select usuario0_.id_usuario as id_usu
```



Recuperando o Usuário Logado



Outra opção é utilizar o objeto `SecurityContextHolder` que contem informações do contexto de segurança, incluindo o usuário autenticado.

Esta opção tem a vantagem de não aumentar o número de parâmetros nos métodos do controller e poder ser utilizada em qualquer parte da aplicação, incluindo nos services criados.

```
@GetMapping
public ResponseEntity<List<UsuarioDTO>> listar(){
    UserDetails details = (UserDetails) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    System.out.println("Login do usuario SecurityContextHolder: " + details.getUsername());

    return ResponseEntity.ok(usuarioService.findAll());
}
```

```
2022-09-07 19:12:21.460 INFO 553169 --- [nio-8080-exec-2] o.s.we
2022-09-07 19:12:21.461 INFO 553169 --- [nio-8080-exec-2] o.s.we
Login do usuario SecurityContextHolder: teste
Hibernate: select usuario0 .id usuario as id usuar1 2 , usuario0
```

CORS e CSRF

Cross-Origin Resource Sharing ou CORS é um mecanismo que permite que recursos restritos em uma página da web sejam recuperados por outro domínio fora do domínio ao qual pertence o recurso que será recuperado

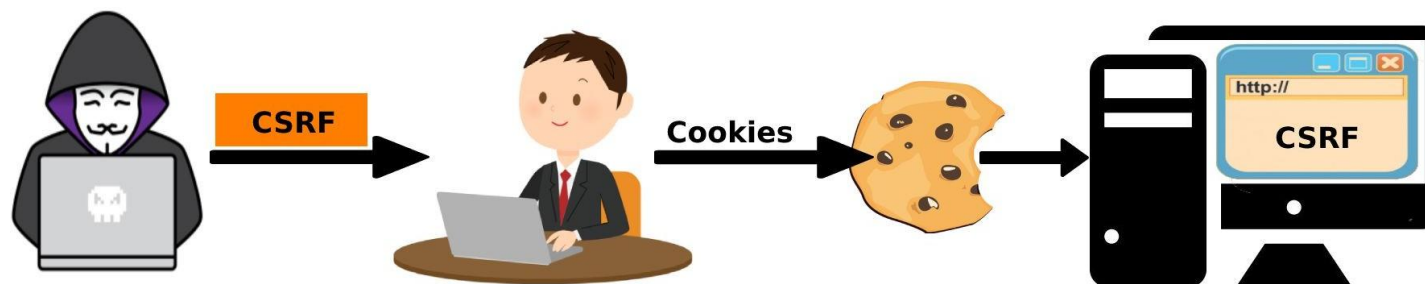
O **cross-site request forgery**, em português: falsificação de solicitações entre sites, também conhecido como ataque de um clique ou montagem de sessão, é um tipo de exploit malicioso de um website, no qual comandos não autorizados são transmitidos a partir de um usuário em quem a aplicação web confia.

Wikipedia

CSRF - Cross Site Request Forgery



É uma vulnerabilidade que forja requisições falsas. é um tipo de ataque que tem como objetivo inserir requisições em sessões que já estejam abertas pelo usuário. O ataque CSRF ocorre quando a vítima executa um script, sem perceber, no seu navegador, e este script explora a sessão iniciada em um determinado site, através de cookies armazenados no navegador.



É conduzido tipicamente com a ajuda da engenharia social, onde o atacante envia um email contendo um link para a vítima. Link esse que ao ser clicado realiza uma requisição forjada para a aplicação web alvo. Como a vítima provavelmente estará autenticada na aplicação alvo na hora do ataque, é impossível que a aplicação web alvo consiga distinguir entre uma requisição legítima de uma requisição forjada. Mais informações em <https://www.infosec.com.br/cross-site-request-forgery/>

CSRF - Habilitando e Desabilitando



CSRF já vem habilitado nas configurações de segurança por padrão no Spring Security, não sendo necessário nenhuma ação para habilitá-lo. É possível verificar no header da resposta o token gerado em cada requisição.

Em aplicações mais modernas, aplicações REST Stateless que usam JWT (Json Web Token - veremos mais a frente), não há a necessidade da utilização de CSRF no Spring Security, neste caso podemos desabilitá-lo da configuração inserindo o código **http.csrf(csrf-> csrf.disable())**.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf-> csrf.disable()).httpBasic(Customizer.withDefaults()).authorizeHttpRequests(requests -> {
        requests.requestMatchers(HttpMethod.GET, "/funcionarios").permitAll();
        requests.requestMatchers(HttpMethod.GET, "/usuarios").hasRole("RH").anyRequest().authenticated();
    }).sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```


CORS



É um recurso utilizado pelos navegadores para compartilhamento de recursos em origens diferentes.

Quando colocamos nossa API em produção podemos encontrar o erro de CORS, quando o Frontend for acessar o backend estando em outro endereço/servidor.

Os navegadores fazem uso de um recurso de segurança chamado **Same-Origin Policy**. Ele serve para limitar como um script de uma origem pode interagir com recursos de outra origem. Um recurso de um site por padrão só pode ser chamado por outro site se os dois sites estiverem sob o mesmo domínio limitando assim chamada de APIs REST por aplicações JS, por exemplo, hospedadas em servidores diferentes. O navegador leva em conta o protocolo (http ou https) , o número da porta e subdomínio.



Postman - não vai ter problema de CORS por não ser uma página que está querendo acessar outro servidor.

Com a implementação do CORS um domínio permite ao outro a comunicação de forma liberada, essa configuração é feita no backend.

Configurando CORS



```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())
        .cors((cors) -> cors.configurationSource(corsConfigurationSource()))
        .httpBasic(Customizer.withDefaults()).authorizeHttpRequests(requests -> {
            requests.requestMatchers(HttpMethod.GET, "/funcionarios").permitAll();
            requests.requestMatchers(HttpMethod.GET, "/usuarios").hasRole("RH").anyRequest().authenticated();
        })
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```

Inserir as configurações de CORS na classe **ConfigSeguranca** depois da configuração do `.csrf()`

Incluir o método que retorna o bean **CorsConfigurationSource**.

```
@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration corsConfiguration = new CorsConfiguration();
    corsConfiguration.setAllowedOrigins(Arrays.asList("http://localhost:3000/"));
    corsConfiguration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "OPTIONS", "HEAD"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", corsConfiguration.applyPermitDefaultValues());
    return source;
}
```

Nele serão declarados as origens permitidas (em desenvolvimento, localhost:3000 geralmente é o endereço do front, em produção deve-se colocar o endereço do servidor correto)

<https://docs.spring.io/spring-security/reference/servlet/integrations/cors.html>

importação
`org.springframework.web.cors`

Testando CORS no Postman



O postman não é a ferramenta ideal para testar CORS, mas é possível verificar se a configuração está funcionando passando o header origin na requisição, com um endereço diferente do que foi configurado:

The screenshot shows a GET request to `http://localhost:8080/funcionarios` in Postman. The 'Headers' tab is active, showing three headers: `Accept-Encoding` (value: `gzip, deflate, br`), `Connection` (value: `keep-alive`), and `origin` (value: `http://localhost:2000`). The status bar at the bottom indicates a `403 Forbidden` response with a 38 ms latency and 430 B body size. The response body is displayed in 'Text' format, showing the message: `1 Invalid CORS request`.

The screenshot shows a GET request to `http://localhost:8080/funcionarios` in Postman. The 'Headers' tab is active, showing three headers: `Accept-Encoding` (value: `gzip, deflate, br`), `Connection` (value: `keep-alive`), and `origin` (value: `http://localhost:3000`). The status bar at the bottom indicates a `200 OK` response with a 25 ms latency and 2.93 KB body size. The response body is displayed in 'JSON' format, showing an array of objects. The first object is:

```
{  "id": 1,  "nome": "Carlos",  "salario": 1000.0,  "dataNascimento": "2000-05-10"}
```



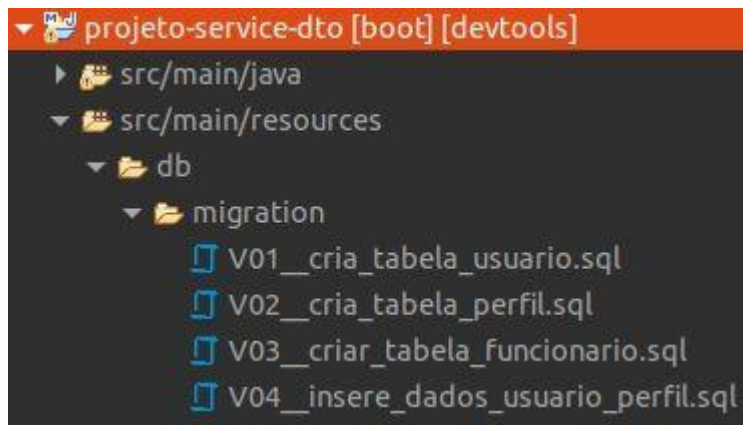
Para usarmos autenticação utilizando o usuário e senha armazenados no banco, teremos que fazer algumas alterações na nossa aplicação.

Usuario e
Senha a partir
do Banco

Preparar a tabela de usuários



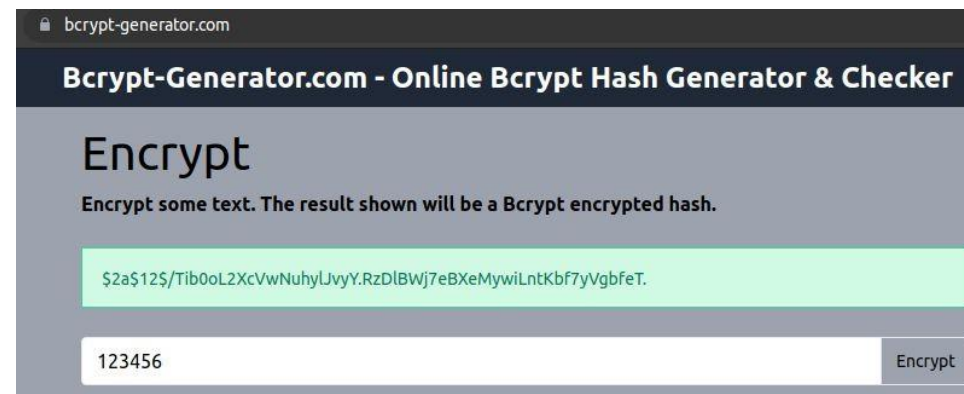
Vamos apagar os dados das tabelas usuários e perfil e inserir alguns usuários com senhas criptografadas (não é uma boa prática salvar senhas diretamente no banco). Para isso vamos criar mais um arquivo de migração.



```
DELETE from perfil_usuario;
DELETE from perfil;
DELETE from usuario;
INSERT INTO usuario (nome, email, senha) VALUES
('Joao da Silva', 'joao@email.com', '$2a$12$sPPV9up/RlaZGUBA1AU7ju66f4o.eNSGhhCaWUdr4rnvDZ.QjaMtK'),
('Andre das covas', 'andre@email.com', '$2a$12$G7ibc/sJRL0BWCpVCBcRxudHZ2aV8uHbMhHbu/Y6Zpz3Dw1X4.B2S');
INSERT INTO perfil (nome) VALUES
('ADMIN'),
('USER');
INSERT INTO perfil_usuario (id_usuario, id_perfil) VALUES
( (SELECT id_usuario FROM usuario WHERE email='joao@email.com'),
  (SELECT id_perfil FROM perfil WHERE nome='ADMIN') ),
( (SELECT id_usuario FROM usuario WHERE email='joao@email.com'),
  (SELECT id_perfil FROM perfil WHERE nome='USER') ),
( (SELECT id_usuario FROM usuario WHERE email='andre@email.com'),
  (SELECT id_perfil FROM perfil WHERE nome='USER') );
```

Senhas criptografadas pelo site <https://bcrypt-generator.com/>

123456	\$2a\$12\$sPPV9up/RlaZGUBA1AU7ju66f4o.eNSGhhCaWUdr4rnvDZ.QjaMtK
654321	\$2a\$12\$G7ibc/sJRL0BWCpVCBcRxudHZ2aV8uHbMhHbu/Y6Zpz3Dw1X4.B2S



Senhas criptografadas no banco



Para configurar o encoder, basta incluir a criação do Bean dentro da classe de configuração de segurança:

```
@Autowired
BCryptPasswordEncoder encoder;
```

```
public UsuarioDTO inserir(UsuarioInserirDTO user) throws EmailException {
    if (!user.getSenha().equalsIgnoreCase(user.getConfirmaSenha())) {
        throw new SenhaException("Senha e Confirma Senha não são iguais");
    }
    if (usuarioRepository.findByEmail(user.getEmail()) != null) {
        throw new EmailException("Email já existente");
    }
    Usuario usuario = new Usuario();
    usuario.setNome(user.getNome());
    usuario.setEmail(user.getEmail());
    usuario.setSenha(encoder.encode(user.getSenha()));
    Set<UsuarioPerfil> perfis = new HashSet<>();
    for(Perfil perfil: user.getPerfis()) {
        perfil = perfilService.buscar(perfil.getId());
        UsuarioPerfil usuarioPerfil = new UsuarioPerfil(usuario, perfil, LocalDate.now());
        perfis.add(usuarioPerfil);
    }
    usuario.setUsuarioPerfis(perfis);
    usuario = usuarioRepository.save(usuario);
    return new UsuarioDTO(usuario);
}
```

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

Para garantir que as novas senhas sejam criptografadas quando um usuário for salvo pelo `UsuarioService`, devemos incluir o encoder como dependência e criptografar a senha antes da inserção no banco.

Esta criptografia é “via de mão única”, ou seja, não é possível, a partir da senha criptografada, descriptografá-la e saber a senha original.

UserDetails



Spring utiliza classes que implementam a interface `UserDetailsService` para recuperar as informações do usuário que está se autenticando. Esta classe deve usar objeto que implementa a interface `UserDetails`, tendo os atributos login do usuário, sua senha e suas “GrantedAuthority” (autoridade concedida - perfis do usuário) .

Vamos fazer a nossa classe **Usuario** implementar **UserDetails**

O método **getAuthorities** deve retornar a lista de perfis do usuário, utilizando a classe **SimpleGrantedAuthority**.

O método **deve retornar o valor que é utilizado como login do usuário. Neste exemplo estamos utilizando o e-mail, mas em alguns caso pode-se ter um atributo login na entidade usuário.**

Já o **getPassword** deve retornar a senha criptografada. Como já estamos armazenando ela criptografada, basta retorná-la.

```
@Entity
public class Usuario implements UserDetails, Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_usuario")
    private Long id;
    private String nome;
    private String email;
    private String senha;
    private static final long serialVersionUID = 1L;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        for(UsuarioPerfil usuarioPerfil : usuario.getUsuarioPerfis()){
            authorities.add(new SimpleGrantedAuthority(usuarioPerfil.getId().getPerfil().getNome()));
        }
        return authorities;
    }

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

    @Override
    public String getUsername() {
        return email();
    }
}
```

Os outros métodos como não vamos utilizar colocar o retorno para **true**.

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

UserDetailsService



Precisamos criar um **serviço** padrão para validação das informações do usuário (login e senha) no **Spring Security**. Para isto precisamos criar uma classe implementando **UserDetailsService**. Lembre-se que estamos utilizando o e-mail do usuário com login (username). O método **loadUserByUsername** vai buscar as informações de login do usuário.

```
@Service
public class UsuarioDetalheImpl implements UserDetailsService{
    @Autowired
    private UsuarioRepository usuarioRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Usuario usuario = usuarioRepository.findByEmail(username);
        if (usuario==null){
            throw new RuntimeException();
        }
        return usuario;
    }
}
```

como usuário implementa **UserDetails** retornamos o usuário

Será utilizado o método **findByEmail** no repositório de usuários

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{
    Usuario findByEmail(String email);
}
```


Configuração de Segurança



Devemos substituir a configuração anterior, que utilizava uma autenticação em memória pelo uso do **UserDetailsService** que implementamos.

OBS: Retirar da classe **ConfigSeguranca** o Bean para autenticação em memória.

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
    UserDetails user = User.withDefaultPasswordEncoder().
        username("teste").
        password("123456").
        roles("RH").build();
    return new InMemoryUserDetailsManager(user);
}
```

Inserir o Bean do **AuthenticationManager** para permitir autenticação personalizada, como em um **@Service** onde procuramos o email do usuário no banco de dados.

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

<https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/index.html#publish-authentication-manager-bean>

Verificando a autenticação



GET ▼ http://localhost:8080/usuarios Send ▼

Params **Authorization** ● Headers (9) Body Pre-request Script Tests Settings Cookies

Type
Basic Auth ▼

Username: teste
Password: 123456
☒ Show Password

The authorization header will be automatically generated when you send the request. [Learn more](#)

Body Cookies (1) Headers (15) Test Results 401 Unauthorized 11 ms 495 B Save Response ▼

Pretty Raw Preview Visualize Text ▼ ≡

1

GET ▼ http://localhost:8080/usuarios Send ▼

Params **Authorization** ● Headers (9) Body Pre-request Script Tests Settings Cookies

Type
Basic Auth ▼

Username: joao@email.com
Password: 123456
☒ Show Password

The authorization header will be automatically generated when you send the request. [Learn more](#)

Body Cookies (1) Headers (15) Test Results 200 OK 608 ms 697 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ ≡

```
1 [
2   {
3     "id": 4,
4     "nome": "Joao da Silva",
5     "email": "joao@email.com",
6     "perfis": [
7       {
8         "id": 1,
9         "nome": "ADMIN"
10      },
11      {
12        "id": 2,
13        "nome": "USER"
```

Permissões de Acesso



O método **requestMatchers** tem diversas opções que permite realizarmos um ajuste fino nas permissões, podendo inclusive definir quais métodos/verbos HTTP e quais perfis podem ter acesso.

```
http.authorizeHttpRequests()  
    .requestMatchers("/", "/public/**").permitAll()  
    .requestMatchers("/funcionarios").permitAll()  
    .requestMatchers("/funcionarios/salarios-por-idade").permitAll()  
    .requestMatchers(HttpMethod.GET, "/funcionarios/salario", "/funcionarios/pagina", "/funcionarios/nome").hasAuthority("ADMIN")  
    .requestMatchers(HttpMethod.GET, "/usuarios").hasAnyAuthority("ADMIN", "USER")  
    .requestMatchers(HttpMethod.POST, "/usuarios").hasAuthority("ADMIN")
```

- a url / e todo o conteúdo de /public pode ser acessado por todos
- /funcionario pode ser acessado por todos, mesmo por pessoas não autenticadas (público)
- /funcionario/salario só pode ser acessado por um usuário ADMIN e apenas no método GET
- /funcionario/pagina só pode ser acessado por um usuário ADMIN e apenas no método GET
- /funcionario/nome só pode ser acessado por um usuário ADMIN e apenas no método GET
- /usuarios pode ser acessado no método GET por usuários com perfis ADMIN ou USER
- /usuarios pode ser acessado no método POST apenas por usuário ADMIN

Ou seja, usuários que tenham o perfil ADMIN podem consultar salários, pagina nome, consultar usuários e incluir novos usuários. Usuários com o perfil USER só podem consultar usuários.

Qualquer pessoa pode consultar funcionários e / (index.html por exemplo) e o conteúdo de public.

Permissões de Acesso



Usuário com perfil apenas USER, não tem acesso: status 403 - proibido acesso

The screenshot shows the Postman interface for a GET request to `http://localhost:8080/funcionarios/salario`. The **Authorization** tab is selected, showing Basic Authentication with the username `andre@email.com` and password `654321`. The **Body** tab is also selected, showing a status of `403 Forbidden` with a response time of `393 ms` and a size of `401 B`. The response body is empty, indicating that the user does not have access.

Usuário com perfil apenas ADMIN, acesso concedido

The screenshot shows the Postman interface for a GET request to `http://localhost:8080/funcionarios/salario`. The **Authorization** tab is selected, showing Basic Authentication with the username `joao@email.com` and password `123456`. The **Body** tab is selected, showing a status of `200 OK` with a response time of `587 ms` and a size of `2.19 KB`. The response body is a JSON array containing two employee records:

```
1 {
2   "content": [
3     {
4       "id": 1,
5       "nome": "Carlos",
6       "salario": 1000.0,
7       "dataNascimento": "2000-05-10"
8     },
9     {
10      "id": 2,
11      "nome": "João",
12      "salario": 2000.0,
13      "dataNascimento": "1999-06-11"
14    }
15  ]
16 }
```

Aplicações tradicionais onde o front-end e back-end estão na mesma aplicação é utilizado o conceito de sessão para autorizarmos os recursos disponíveis na aplicação. Em aplicações REST são utilizados tokens para liberar os recursos.

JWT:

JWT é um padrão que tem o objetivo de transmitir mensagens de uma forma segura utilizando um token compacto no formato de um objeto JSON.

Json

Web

Token

O JWT permite autenticar um usuário e garantir que as demais requisições serão feitas de forma autenticada, sendo possível restringir acessos a recursos e serviços com diferentes níveis de permissões.

Estrutura JWT



Um token JWT consiste em três partes separadas por pontos.

Cada parte é uma String Base64-URL

Cada string representa uma parte do token:

- Header
- Payload
- Signature

O site <https://jwt.io> exibe a estrutura de forma detalhada:

The screenshot displays the JWT.io interface. On the left, under the 'Encoded' tab, a token is pasted: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.QoJTv2PltzAigRvprj8xufd8bVn_KS1880I8x7AqQ_g`. On the right, the 'Decoded' tab shows the token's structure. The 'HEADER: ALGORITHM & TOKEN TYPE' section contains `{ "alg": "HS256", "typ": "JWT" }`. The 'PAYLOAD: DATA' section contains `{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }`. The 'VERIFY SIGNATURE' section shows the HMACSHA256 function being applied to the base64 encoded header and payload, with a checkbox for 'secret base64 encoded' checked.

Estrutura JWT



O header ou cabeçalho normalmente consiste em duas partes: o algoritmo de assinatura que está sendo utilizado e o tipo de token JWT.

A segunda parte é o **payload** ou corpo, que contém as **claims** que normalmente são informações do usuário autenticado.

A chave sub é obrigatória, pois a mesma é um identificador da entidade a qual o token se refere.

HEADER: ALGORITHM & TOKEN TYPE
<pre>{ "alg": "HS256", "typ": "JWT" }</pre>
PAYLOAD: DATA
<pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }</pre>

Claims reservados

Não são obrigatórios, mas importantes para a segurança da API.

sub - Entidade a quem o token pertence, normalmente o login do usuário (ou o id).

iss - identifica o emissor do token.

exp - Tempo de quando o token irá expirar.

iat - quando o token foi criado.

Estrutura JWT



A assinatura é a concatenação dos **hashes** gerados a partir do **Header** e **Payload** usando base64UrlEncode. As hashes são criadas a partir do algoritmo de assinatura indicado.

A assinatura é utilizada para garantir a integridade do token, prevenindo ataques de interceptação man-in-the-middle, pois se o invasor modificar o conteúdo do token, o hash final não será válido pois não foi assinado com sua chave secreta. Apenas quem está de posse da chave pode criar, alterar e validar o token.

Esta chave geralmente é parte da configuração da aplicação, ficando no servidor e não sendo enviada para o frontend.

Vamos incluí-la como uma variável de aplicação no nosso arquivo de configuração, bem como um tempo de expiração para o token.

VERIFY SIGNATURE


```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  jhjj  
) ☒ secret base64 encoded
```

```
spring.datasource.url=jdbc:postgresql://localhost:5432/projeto  
spring.datasource.username=postgres  
spring.datasource.password=postgres  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=none  
spring.jackson.deserialization.fail-on-unknown-properties=false  
  
auth.jwt-secret=EAssimQueMeuFuscaAnda_EAssimQueEleVaiParar  
auth.jwt-expiration-miliseg=120000
```


Dependencia JWT



Home » io.jsonwebtoken » jjwt-api » 0.11.5

 **JJWT :: API » 0.11.5**
JJWT :: API

License	Apache 2.0
Categories	JWT Libraries
Tags	api jwt json security
Date	Apr 28, 2022
Files	pom (1 KB) jar (75 KB) View All
Repositories	Central
Ranking	#1095 in MvnRepository (See Top Artifacts) #3 in JWT Libraries
Used By	375 artifacts

Maven [Gradle](#) [Gradle \(Short\)](#) [Gradle \(Kotlin\)](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
```

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
```

```
</dependency>
```

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
```

```
</dependency>
```

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-gson</artifactId>
  <version>0.11.5</version>
```

```
</dependency>
```


JwtUtil



Agora vamos criar uma classe **JwtUtil** no pacote security, responsável por gerar o token a partir do username (no nosso caso, email do usuário), usando um determinado algoritmo de criptografia. Usaremos a anotação **@Component**, que tem o mesmo efeito de **@Service** ou **@Controller**, indicar que ela é um componente para o Spring e pode ser “injetada” em outras classes

Pegar os valores definidos no arquivo de propriedades e inserir nos atributos **jwtSecret** e **jwtExpirationMilliseg**.

```
@Component
public class JwtUtil {

    @Value("${auth.jwt-secret}")
    private String jwtSecret;
    @Value("${auth.jwt-expiration-milise
g}")
    private Long jwtExpirationMilliseg;

    public String generateToken(String username) {
        SecretKey secretKeySpec =
            Keys.hmacShaKeyFor(jwtSecret.getBytes());
        return Jwts.builder()
            .setSubject(username)
            .setExpiration(new Date(System.currentTimeMillis() +
                this.jwtExpirationMilliseg))
            .signWith(secretKeySpec)
            .compact();
    }
```

Claims - armazena as informações do usuário

```
public boolean isValidToken(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;
        }
    }
```

O método **builder** é o responsável pela geração do token, o **Subject** é o nome do usuário, o **Expiration** é o tempo de expiração do token que pega o horário atual do sistema mais o tempo definido no atributo **jwtExpirationMisiseg** o método **signWith** é o algoritmo de assinatura passando o atributo **jwtSecret** como argumento.

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

public Claims getClaims(String token) {
    return
        Jwts.parserBuilder().setSigningKey(secretKeySpec).build
            ().parseClaimsJws(token).getBody();
}
```

LoginDTO



Para realizarmos a autenticação com token, usaremos um endpoint padrão do Spring Security (/login). Nele passaremos um json contendo o usuário e a senha (formulário de login padrão) . Para isso vamos precisar de um DTO para este JSON.

Um objeto simples, contendo dois atributos (username e password) e os respectivos gets e sets

```
public class LoginDTO {  
    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;  
    }  
}
```

JwtAuthenticationFilter



Criação e validação do token JWT

Vamos criar a classe **JwtAuthenticationFilter** no pacote **security**, um filtro de autenticação que ao realizar uma requisição de login, ele vai interceptar e fazer a autenticação. Ela deverá herdar da classe **UsernamePasswordAuthenticationFilter**. O método **attemptAuthentication** é quem lida com a tentativa de autenticação. Pegamos o username e password da requisição, e utilizamos o **AuthenticationManager** para verificar se os dados são correspondentes aos dados do nosso usuário existente.

```
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 {
            LoginDTO login = new ObjectMapper().readValue(request.getInputStream(), LoginDTO.class);
            UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(login.getUsername(), login.getPassword(), new ArrayList<>());
            Authentication auth = authenticationManager.authenticate(authToken);
            return auth;
        } catch (IOException e) {
            throw new RuntimeException("Falha ao autenticar usuário",e);
        }
    }

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

O **AuthenticationManager** é usado pelo nosso filter para autenticação de usuários.

importações de Authentication
`import org.springframework.security.core`

JwtAuthenticationFilter - or partes



O **AuthenticationManager** é usado pelo nosso filter para autenticação de usuários.

Quando criamos uma classe herdando de **UsernamePasswordAuthenticationFilter** o Spring Security sabe que deverá ser feita uma interceptação de login através do endpoint padrão do Spring Security (<http://localhost:8080/login>)

```
public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter{  
    private AuthenticationManager authenticationManager;  
    private JwtUtil jwtUtil;  
    public JwtAuthenticationFilter(AuthenticationManager authenticationManager, JwtUtil jwtUtil){  
        this.authenticationManager=authenticationManager;  
        this.jwtUtil=jwtUtil;  
    }  
}
```

JwtAuthenticationFilter - por partes



Vamos pegar os dados do objeto da requisição através do `HttpServletRequest`, vamos instanciar `UsernamePasswordAuthenticationToken` com os dados da requisição, a partir do objeto armazenado na variável `authToken` nós vamos "chamar" o método `authenticate` que vai fazer a verificação se os dados do usuário são válidos e tem como retorno um objeto do tipo `Authentication`.

```
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws
AuthenticationException{

    try {
        LoginDTO login = new ObjectMapper().readValue(request.getInputStream(), LoginDTO.class);
        UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(login.getUsername(),
login.getPassword(), new ArrayList<>());
        Authentication auth = authenticationManager.authenticate(authToken);
        return auth;
    } catch (IOException e) {
        throw new RuntimeException("Falha ao autenticar usuário",e);
    }
}
```

JwtAuthenticationFilter - por partes



Quando o usuário for autenticado com sucesso, o método irá retornar um JWT com a autorização Authorization no cabeçalho da resposta.

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

JwtAuthorizationFilter



Criamos a classe **JWTAuthorizationFilter** no pacote **security**, ou seja um filtro de autorização que ao você fazer uma requisição ele vai interceptar e verificar se o token enviado é válido. Ela extend da classe **BasicAuthenticationFilter**. Implementamos o método **doFilter** para ao fazer a requisição verificar a validade do token;

```
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;
    }
    @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.isValidToken(token)) {
            String username = jwtUtil.getUsername(token);
            UserDetails user = userDetailsService.loadUserByUsername(username);
            return new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
        }
        return null;
    }
}
```

JwtAuthorizationFilter - por partes



Vamos buscar no banco de dados através do [UserDetailsService](#) para ver se o usuário existe.

```
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 - por partes



Método que intercepta a requisição e verifica se o usuário está autorizado

```
@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);
    }
}
```

Pegamos o valor que vem no cabeçalho da requisição através do método `request.getHeader`

O método `getAuthentication` recebe o token como argumento aí devemos retirar a palavra "Bearer " para pegarmos somente valor do token.

Se tudo estiver ok vamos chamar o método `getContext` para liberar a autorização do usuário

JwtAuthorizationFilter - por partes



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

se o token estiver válido vamos pegar o nome do usuário dentro do token

Configuração de Segurança



Vamos adicionar o filtro na classe **ConfigSeguranca**.
http.addFilter

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable()).cors((cors) -> cors.configurationSource(corsConfigurationSource()))
    .authorizeHttpRequests(requests ->
        requests.requestMatchers("/public/**").permitAll().
        requestMatchers("/funcionarios").permitAll().
        requestMatchers("/funcionarios/salarios-por-idade").permitAll().
        requestMatchers(HttpMethod.GET, "//funcionarios/salario", "/funcionarios/pagina", "/funcionarios/nome").hasAuthority("ADMIN").
        requestMatchers(HttpMethod.GET, "/usuarios").hasAnyAuthority("ADMIN", "USER").
        requestMatchers(HttpMethod.POST, "/usuarios").hasAuthority("ADMIN")
        .anyRequest().authenticated()
    ).sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    http.addFilter(new JwtAuthenticationFilter(
        authenticationManager(http.getSharedObject(AuthenticationConfiguration.class)), jwtUtil));
    http.addFilter(new JwtAuthorizationFilter(
        authenticationManager(http.getSharedObject(AuthenticationConfiguration.class)), jwtUtil, userDetailsService));
    return http.build();
}
```

Realizando a Autenticação



O endpoint `/login` é padrão do Spring, por isso não precisamos implementar o **Controller** para esse endpoint. O Spring vai usar a implementação que fizemos em `UserDetails` e `UserDetailsService` para validar o usuário e senha e devolver o token. Para autenticação, vamos enviar uma requisição do tipo **POST** para o endereço `http://localhost:8080/login` com as credencias (LoginDTO) do nosso usuário no corpo da requisição.

```
{
  "username": "joao@email.com"
  "password": "123456"
}
```

No cabeçalho da resposta dessa requisição temos o nosso token com o prefixo Bearer.

POST http://localhost:8080/login

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "username": "joao@email.com",
3   "password": "123456"
4 }
```

Body Cookies (1) Headers (15) Test Results 200 OK 761 ms 584 B Save Response

Header	Value
Origin	Vary
Access-Control-Request-Method	Vary
Access-Control-Request-Headers	Vary
Authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqb2FvQGVTYWIsLmNvbSIsImV4cCI6MTY2...
access-control-expose-headers	Authorization
X-Content-Type-Options	nosniff
X-XSS-Protection	1; mode=block
Cache-Control	no-cache, no-store, max-age=0, must-revalidate

Passando a Autenticação na requisição



Temos que passar no header da requisição, o cabeçalho Authorization com o valor "Bearer ..." contendo o token que recuperamos na resposta do login.

Podemos utilizar a aba Authorization do POSTMAN para facilitar os testes.

