Programação Backend com Spring boot

Por Manoel C M Neto

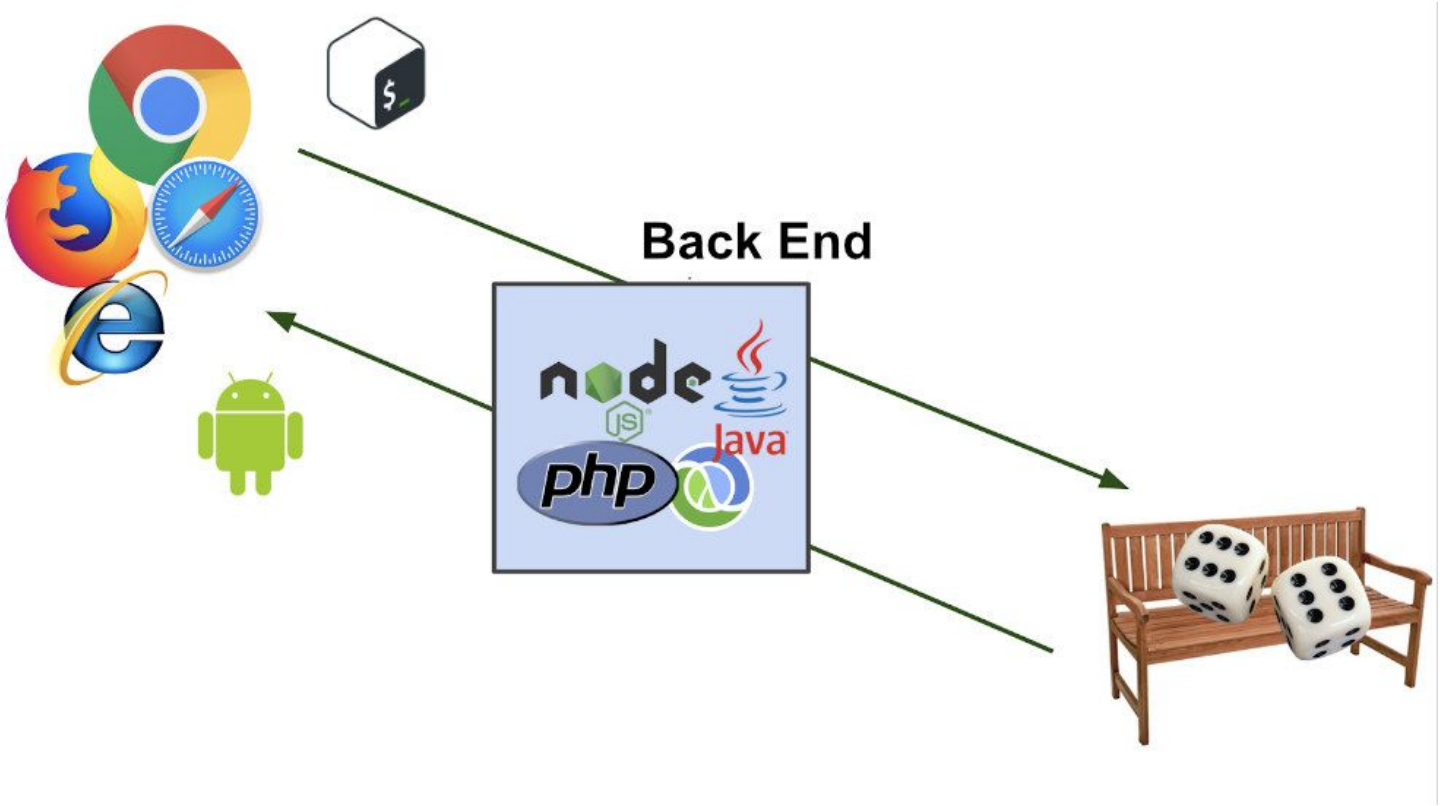
Front end x Back end na Web

- O **Front End** é a parte visual de um site, aquilo que conseguimos interagir.
- Quem trabalha com Front End é responsável por desenvolver por meio de código uma interface gráfica.
- Normalmente usamos as tecnologias base da Web (HTML, CSS e JavaScript).
- Aqui vamos fazer isso com React Js
- **Back End**, como o próprio nome sugere, vem da ideia do que tem por trás de uma aplicação.
- Para conseguir usar o Instagram, os dados do seu perfil, amigos e publicações precisam estar salvos em algum banco de dados e processados a partir de lá.

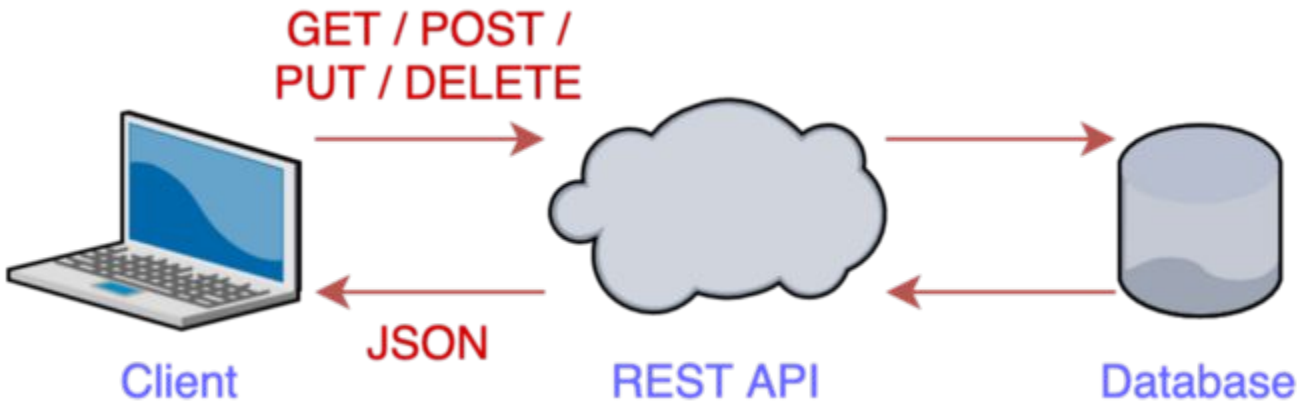
Front end x Back end na Web

- **Back End**, faz a ponte entre o Front e os serviços usados em um sistema como: banco de dados, sistemas de pagamento, envio de email, etc.
- É responsabilidade do Back end aplicar as devidas regras de negócio, validações e garantias em um ambiente onde o usuário final não tenha acesso e possa manipular algo.
- E um dev **Full Stack**? Faz os dois!

Front end x Back end na Web



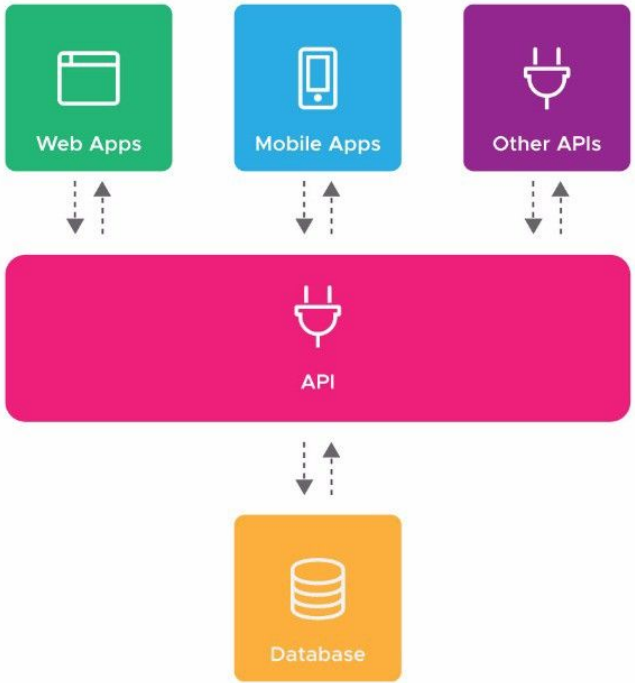
Arquitetura REST (Representational State Transfer)



Arquitetura REST (Representational State Transfer)

- Formato JSON(Javascript Object Notation)
- https://sujeitoprogramador.com/r-api/?api=filmes

Arquitetura REST (Representational State Transfer)



Entendendo uma requisição para uma API REST

- **Endpoint**: A URL nada mais é que o caminho para fazer a requisição, porém é interessante ressaltar que ela segue a seguinte estrutura:
  - Base URL. Ex: https://sujeitoprogramador.com/
  - Resource ou Path. O recurso é o tipo de informação que você está buscando. No nosso exemplo são filmes. https://sujeitoprogramador.com/r-api/?api=filmes
- **Métodos**: O método te ajuda a informar o tipo de ação que você está fazendo naquela requisição. São os principais "verbos" do HTTP:
  - **Get** (Buscar dados)
  - **Post** (Enviar dados)
  - **Put** e **Patch** (Atualizar dados)
  - **Delete** (Deletar dados)

Entendendo uma requisição para uma API REST

- **Headers**: Headers ou cabeçalhos permitem que você envie informações adicionais na requisição. Ele pode ser utilizado para inúmeras funções, como: autenticação, formatação de objeto, e muito mais.
- **Body:** O body é o corpo da mensagem que você quer enviar na requisição. Ele é utilizado somente nos métodos de POST, PUT, PATCH, ou seja, ele contém o dado a ser processado pela API, e por isso ele não é necessário em métodos de leitura de dados
- **HTTP Status Codes:** Para facilitar o entendimento das respostas das APIs existem padrões de códigos de status que podem ser utilizados
    - 200 (OK), o 201 (created), o 204 (no content), o 404 (not found), o 400 (bad request), e 500 (internal server error).
    - Podemos testar cada verbo na https://reqres.in/

Spring boot

- Spring é um dos frameworks mais antigos do Java e até hoje é muito popular
- Desenvolvido por Rod Johnson entre 2000/2002.
- Ná época usava-se muito o J2EE (JEE) para aplicações corporativas
- Isso envolvia tecnologias complexas como RMI, EJB, etc.
- Rod era um especialista em J2EE e sabia como ninguém montar aplicações que fossem robustas, escaláveis e de fácil manutenção.
- Ele escreveu um livro "Expert One-on-One: J2EE Design and Development".
- No livro ele apresentou uma biblioteca "alternativa" ao J2EE.
- Essa biblioteca foi disponibilizada para download e posteriormente foi chamada de Spring Framework.

Spring boot

- A ideia do Spring era ser uma alternativa a esse modelo complexo do J2EE.
- O grande foco dele era a simplicidade de código.
- E isso agradou muito os desenvolvedores, que conseguiram implementar seu sistema com classes Java seguindo o padrão POJO, classe simples, sem ter muita dependência de infraestrutura, nem nada muito complexo.
- Depois de um tempo, com novos frameworks como JSF o Spring "esfriou".
- Porém, em 2013, 2014, o pessoal do Spring criou o **Spring Boot**, que foi um projeto que revolucionou o desenvolvimento para Java e que fez o Spring alavancar de novo no mercado

Spring boot

- A ideia do Spring Boot é que você consiga desenvolver uma aplicação sem o uso de um container.
- Temos um servidor embutido e rodamos a aplicação em um método main.
- Posso gerar o build da minha aplicação como sendo um JAR, que é muito mais leve e mais simples de ser executado que um WAR gerando em aplicações web tradicionais em Java.
- A ideia do Spring Boot é que muitas coisas já vêm configuradas por padrão para você.
- Um grande "case" é o da Netflix, que tem vários projetos que o Spring Boot simplificou muito.

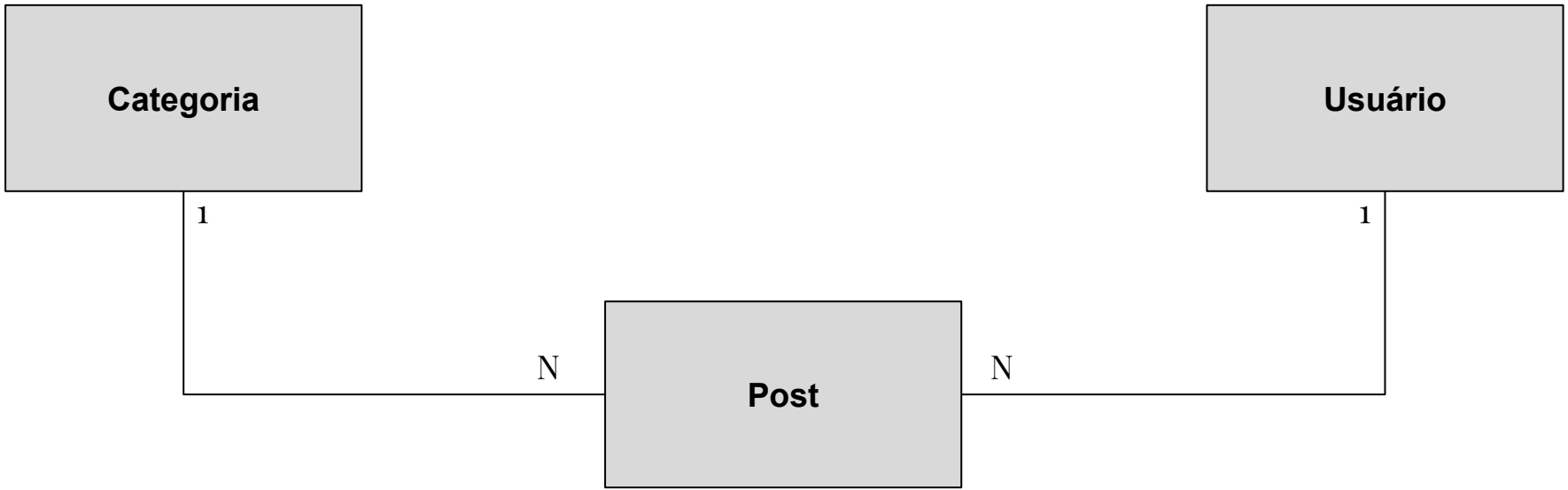
Spring boot: Criando um projeto e rodando o Hello World

- A primeira coisa que precisamos fazer é criar nosso projeto.
- Spring criou um site que é um gerador de projetos com Spring Boot, chamado Spring initializr. https://start.spring.io/ (USE A **VERSÃO 3.3.5)**
- Cria uma classe HelloController em um pacote controller dentro do pacote principal.
- Anote com @Controller antes da declaração de classe.
- Crie um método que retorna uma String chamado hello() e retorne "Hello Info12!"
- Anote o método com @RequestMapping("/helloword") e @ResponseBody
- Execute a classe principal e abra um navegador digitando :
    - http://localhost:8080/helloworld

Spring boot: Criando um projeto e rodando o Hello World

```
1.  @Controller
2.  public class HelloController {
3.
4.      @RequestMapping("/helloworld")
5.      @ResponseBody
6.      public String hello() {
7.          return "Hello INF012!";
8.      }
9.
10. }
```

Projeto blog de notícias



Classes de Modelo

```
1.  public class Usuario {
2.      private Long id;
3.      private String nome;
4.      private String login;
5.      private String senha;
6.
7.      public String getNome() {
8.          return nome;
9.      }
```

```
1.  public void setNome(String nome) {
2.          this.nome = nome;
3.      }
4.      public String getLogin() {
5.          return login;
6.      }
7.      public void setLogin(String login) {
8.          this.login = login;
9.      }
```

Classes de Modelo

```
1.  public enum Categoria {
2.
3.          POLITICA,
4.          ESPORTE,
5.          EDUCACAO,
6.          COTIDIANO,
7.          FOFOCAS;
8.  }
```

Classes de Modelo

```
1.  public class Post {
2.
3.      private Long id;
4.      private String titulo;
5.      private String texto;
6.      private Usuario usuario
7.      private Categororia categoria;
8.
9.      public Long getId() {
10.         return id;
11.     }
12.
```

```
public void setTitulo(String titulo) {
        this.titulo = titulo;
}
public String getTexto() {
        return texto;
}
public void setTexto(String texto) {
        this.texto = texto;
}
```

Criando um controller REST para o blog

```
1.  @RestController
2.  public class PostController {
3.  @RequestMapping("/posts")
4.          public Post listar() {
5.                  Post post=new Post();
6.                  post.setTitulo("Titulo de exemplo")
7.             return post;
8.          }
9.  }
```

DICA: Atualizando o Servidor automaticamente !

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
</dependency>
```

Spring Data: Usando um banco de dados

- Até aqui, usamos dados em memória. No mundo real os dados viriam de um banco de dados!
- O spring nos ajuda a abstrair ao máximo o uso de banco de dados com uma implementação do JPA (Java Persistence API). Por padrão ele usa **Hibernate.**
- Para isso precisamos incluir essa dependência no pow.xml.
- <dependency>
- <groupId>org.springframework.boot</groupId>
- <artifactId>spring-boot-starter-data-jpa</artifactId>
- </dependency>

Spring Data: Usando um banco de dados

- Agora precisamos escolher um banco de dados para usar. Você pode usar qualquer um (MySql, SqlServer, Postgres, etc.). Aqui vamos usar Postgres.
- add no pow.xml:
- <dependency>
- <groupId>org.postgresql</groupId>
- <artifactId>postgresql</artifactId>
- <scope>runtime</scope>
- </dependency>
- Acesse a console do banco usando pgadmin ou outro cliente da sua escolha

Spring Data: No diretório resources edite o application.properties

1. spring.datasource.url=jdbc:postgresql://localhost:5432/**postgres**
2. spring.datasource.username=**postgres**
3. spring.datasource.password=**postgres**
4. spring.datasource.driver-class-name=org.postgresql.Driver
5. spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
6. spring.jpa.hibernate.ddl-auto=update
7. spring.jpa.show-sql=true

Spring Data: Anotando os models para o padrão JPA

```
1. @Entity(name="usuarios")
2. public class Usuario {
3.         @Id
4.         @GeneratedValue(strategy = GenerationType.IDENTITY)
5.         private Long id;
6.         private String nome;
7.         private String login;
8.         private String senha;
```

Spring Data: Anotando os models para o padrão JPA

```
1.  @Entity(name="posts")
2.  public class Post {
3.      @Id
4.      @GeneratedValue(strategy = GenerationType.IDENTITY)
5.      private Long id;
6.      private String titulo;
7.      private String texto;
8.      @ManyToOne
9.      private Usuario usuario;
10.     @Enumerated(EnumType.STRING)
11.     private Categoria categoria = Categoria.POLITICA;
```

Spring Data: Cadastre alguns dados na suas tabelas

1. INSERT INTO USUARIOS(nome, login, senha) VALUES('Aluno', 'aluno@email.com', '123456');
2. INSERT INTO USUARIOS(nome, login, senha) VALUES('Manoel', 'manoelnetom@gmail.com', '123456');

3. INSERT INTO POSTS(titulo, texto, usuario_id, categoria) VALUES('Dúvida', 'Erro ao criar projeto', 1, 'EDUCACAO');
4. INSERT INTO POSTS(titulo, texto, usuario_id, categoria) VALUES('Pesquisa Eleitoral', 'Divulgada nova pesquisa', 2, 'POLITICA');

Spring Data: Criando um Repository

```
1. public interface PostRepository extends JpaRepository<Post, Long>{
2.
3. }
```

Usando o repository no controller REST para o blog

```
1.  @RestController
2.  public class PostController {
3.          @Autowired
4.          private PostRepository repository;
5.
6.          @RequestMapping("/posts")
7.          public List<Post> listar() {
8.                  return repository.findAll();
9.          }
10. }
```

Não devemos usar um Model como retorno REST!!! Exemplo:

```
{
    "id": 1,
    "titulo": "Dúvida",
    "texto": "Erro ao criar projeto",
    "usuario": {
      "id": 1,
      "nome": "Aluno",
      "login": "aluno@email.com",
      "senha": "123456"
    },
    "categoria": "EDUCACAO"
  },
```

Criando um Digital Transfer Object (DTO)

```
1.  public class PostDto {
2.      private Long id;
3.      private String titulo;
4.      private String texto;
5.      private String usuario;
6.      private Categoria categoria;
7.
8.      public PostDto(Post post) {
9.          this.id = post.getId();
10.         this.titulo = post.getTitulo();
11.         this.texto = post.getTexto();
12.         this.usuario = post.getUsuario().getNome();
13.         this.categoria = post.getCategoria();
14.     }
15.
16.     public static List<PostDto> converte(List<Post> lista){
17.         return lista.stream().map(PostDto::new).collect(Collectors.toList());
18.     }
```

Alterando o controller REST para usar um DTO

```
1.  @RestController
2.  public class PostController {
3.          @Autowired
4.          private PostRepository repository;
5.
6.          @RequestMapping("/posts")
7.          public List<PostDto> listar() {
8.                  return PostDto.converte(repository.findAll());
9.          }
10. }
```

Retorno REST com DTO. Exemplo:

```
{
    "id": 1,
    "titulo": "Dúvida",
    "texto": "Erro ao criar projeto",
    "usuario": "Aluno",
    "categoria": "EDUCACAO"
  },
  {
    "id": 2,
    "titulo": "Pesquisa Eleitoral",
    "texto": "Divulgada nova pesquisa",
    "usuario": "Manoel",
    "categoria": "POLITICA"
  }
```

Consulta de dados em um Repository com filtros

- Até aqui nós usamos a consulta básica de um repository.
- A consulta com o .findAll() retorna TODOS os registros de uma entidade (tabela).
- Mas existe a possibilidade de **filtrar uma consulta** usando os atributos da entidade de forma MUITO simples.
- Para isso basta declarar métodos na classe repository que sigam padrões de nomenclatura.
- Por exemplo, um método de busca pelo atributo título da Classe Post seria:
    - public List<Post> findBy**Titulo**(String titulo)

Adicioinado um filtro em um Repository

```
1. public interface PostRepository extends JpaRepository<Post, Long>{
                public List<Post> findByTitulo(String titulo);
2. }
```

Alterando o controller REST para usar um filtro

```
1.  @RestController
2.  public class PostController {
3.          @Autowired
4.          private PostRepository repository;
5.
6.          @RequestMapping("/posts")
7.          public List<PostDto> listar() {
8.                  return PostDto.converte(repository.findByTitulo("Dúvida"));
9.          }
10. }
```

Retorno REST com filtro por Título

```
{
    "id": 1,
    "titulo": "Dúvida",
    "texto": "Erro ao criar projeto",
    "usuario": "Aluno",
    "categoria": "EDUCACAO"
  },

```

Alterando o controller para passar parâmetro

```
1.  @RestController
2.  public class PostController {
3.          @Autowired
4.          private PostRepository repository;
5.
6.          @RequestMapping("/posts")
7.          public List<PostDto> listar(String titulo) {
8.                  return PostDto.converte(repository.findByTitulo(titulo));
9.          }
10. }
11. acesse http://localhost:8080/posts?titulo=Dúvida
```

Adicioinado um filtro de Usuario em um Repository de Post

1. public interface PostRepository extends **JpaRepository**<**Post**, **Long**>{

   public List<Post> findByTitulo(String titulo);

   public List<Post> findBy**UsuarioNome**(String nome);

2. }

Vai buscar pelo atributo nome de um Usuario de Post

Alterando o controller para buscar pelo usuario parâmetro

```
1.  @RestController
2.  public class PostController {
3.          @Autowired
4.          private PostRepository repository;
5.
6.          @RequestMapping("/posts")
7.          public List<PostDto> listar(String usuario) {
8.                  return PostDto.converte(repository.findByUsuarioNome(usuario));
9.          }
10. }
11. acesse http://localhost:8080/posts?usuario=Manoel
```

Adicioinado um filtro de busca tipo LIKE

1. public interface PostRepository extends **JpaRepository**<**Post**, **Long**>{

        public List<Post> findByTitulo(String titulo);

        public List<Post> findByUsuarioNome(String nome);

        public List<Post> **findByTituloContaining**(String titulo);

2. }

Vai buscar usando uma query com restrição Like %titulo% de Post

Adicioinado um filtro de busca tipo StartsWith

1. public interface PostRepository extends **JpaRepository**<**Post**, **Long**>{

        public List<Post> findByTitulo(String titulo);

        public List<Post> findByUsuarioNome(String nome);

        public List<Post> findByTituloContaining(String titulo);

        public List<Post> **findByTituloStartsWith**(String titulo);

2. }

Vai buscar usando uma query com restrição Like titulo% de Post

Mais sobre filtros de Busca

- Outros filtros:
  a. List<Movie> findByDirector**EndsWith**(String director);
  b. List<Movie> findByTitleContaining**IgnoreCase**(String title);
  c. List<Movie> findByRating**Not**Containing(String rating);
- Se ainda assim precisamos de uma Query específica podemos usar a anotação @Query:

  @Query("SELECT m FROM Movie m WHERE m.title LIKE %**:title**%")

  List<Movie> searchByTitleLike(**@Param("title")** String title);

- Veja mais em : https://www.baeldung.com/spring-jpa-like-queries
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject

Como cadastrar dados usando o verbo http POST

- Até agora, nós só listamos/consultamos dados do banco. Como cadastrar?
- Primeiro é importante criar um novo método em nosso PostController e modificar o endpoint para o controller responder a "/posts" tanto para listar quanto para cadastrar um novo post.
- Basta mover o @RequestMapping("/posts") para antes da declaração de classe.
- Assim o controller todo passa a responder pelo mesmo endpoint
- Mas como diferenciar os métodos de listar e cadastrar?
- Usar os verbos Http com as anotações: **@GetMapping** e **@PostMapping**

Alterando o controller para usar Get e Post no mesmo endpoint

```
1.  @RestController
2.  @RequestMapping("/posts")
3.  public class PostController {
4.          @Autowired
5.          private PostRepository repository;
6.
7.  @GetMapping
8.          public List<PostDto> listar(String usuario) {
9.                  return PostDto.converte(repository.findByUsuarioNome(usuario));
10.         }
11. @PostMapping
12.         public void cadastrar(PostDto post) {
13.
14.         }
15. }
16. acesse http://localhost:8080/posts?usuario=Manoel
```

Foco no método Post : parâmetro com @RequestBody

```
1.  @Autowired
2.      private UsuarioRepository userRepository;
3.  @PostMapping
4.      public void cadastrar(@RequestBody PostDto postDto) {
5.          Post post= postDto.converte(userRepository)

            repository.save(post);

6.      }
7.  }
```

8. **@RequestBody:** É preciso avisar ao spring que os dados vêm no corpo de uma requisição e não em uma url como no método listar (@GetMapping)

Método convert de PostDto

```
1.  public Post converter(UsuarioRepository userRepositorio) {
2.              // TODO Auto-generated method stub]
3.              Usuario user=userRepositorio.findByNome(usuario);
4.              Post post=new Post(titulo, texto, user );
5.              return post;
6.  }
```

Método Post : retornando código 201 (Created)

```
1.  @Autowired
2.      private UsuarioRepository userRepository;
3.  @PostMapping
4.      public ResponseEntity<PostDto> cadastrar(@RequestBody PostDto postDto,
    UriComponentsBuilder uriBuilder) {
5.          Post post= postDto.converter(userRepository);
6.          repository.save(post);
7.          URI uri=uriBuilder.path("/posts/{id}").buildAndExpand(post.getId()).toUri();
8.          return ResponseEntity.created(uri).body(new PostDto(post));
9.      }
10.
```

Validação: Usando o javax.validation

- Inclua essa dependência no pow.xml
- <dependency>
- <groupId>org.springframework.boot</groupId>
- <artifactId>spring-boot-starter-validation</artifactId>
- </dependency>

Validação: Usando o javax.validation

- Anote os DTOs com as validaçoes desejadas: @NotNull, @NotBlank, @Length, etc.
- public class UsuarioForm {
- private Long id;
-
- **@NotNull(message = "O nome não pode ser nulo")**
- private String nome;
- **@NotBlank(message = "O login não pode ser vazio")**
- **@Length(min=5,message = "O login tem min de 5 caractres")**
- private String login;

Validação: Usando o javax.validation

- Anote os o método POST com @Valid
- @PostMapping
- public ResponseEntity<Usuario> cadastrar(@RequestBody **@Valid** UsuarioForm usuarioForm, UriComponentsBuilder builder) {
- Usuario usuario= usuarioForm.converter();
- repository.save(usuario);
- URI uri = builder.path("/usuarios/{id}").buildAndExpand(usuario.getId()).toUri();
- return ResponseEntity.created(uri).body(usuario);
- }

Put: Atualizar um registro usnado PUT

- **@PutMapping("/{id}")**
- **@Transactional**
- public ResponseEntity<UsuarioForm> atualizar(**@PathVariable Long id**, @RequestBody @Valid UsuarioForm usuarioForm) {
- Usuario usuario = **usuarioForm.atualiza(repository, id)**;
- return new ResponseEntity<UsuarioForm>(new UsuarioForm(usuario),HttpStatus.OK);
- }

Put: Método atualiza em UsuarioForm

```
public Usuario atualiza(UsuarioRepository repository, Long id) {
        Usuario usuario=repository.getById(id);
        usuario.setNome(nome);// nome é atributo de UsuarioForm
        usuario.setLogin(login); // login é atributo de UsuarioForm
        usuario.setSenha(senha);// senha é atributo de UsuarioForm
        return usuario;
    }
```

Deletar: Método deletar usando verbo DELETE do HTTP

```
@DeleteMapping("/{id}")
    @Transactional
    public ResponseEntity<?> deletar(@PathVariable Long id) {
        repository.deleteById(id);
        return new ResponseEntity<>(HttpStatus.OK);
    }
```

Tente Criar, Atualizar e Deletar um objeto Post

- Repita o processo e crie o CRUD de um objeto Post.
- Crie separadamente: Cadastrar, Alterar e Apagar
- Use os verbos POST, PUT e DELETE do HTTP

Paginação

- Até aqui, nosso método de lista posts traz todos os registros de um banco de dados.
- Nós só temos 3 registros....mas como seria se tivéssemos 1000.000?
- A solução natural para isso é paginar ! Trazer os registros aos poucos...
- O Spring boot facilita muito o nosso trabalho quando o assunto é paginação!
- Esse é o método que retorna atualmente todos os registos de Post
- @GetMapping
- public List<PostDto> listar(String titulo){
-
- if((titulo!=null) && (!titulo.equals(""))) {
- return PostDto.converte(repository.findByTitulo(titulo));
- }
-
- return PostDto.converte(repository.findAll());
- }

Paginação

- Para paginar, o primeiro passo é passar o **número da página** e **quantidade** de registros por página
- O @RequestParam define que são parâmetros passados via url que podem ou não ser obrigatórios (required = false)
- @GetMapping
- public List<PostDto> listar(@RequestParam(required = false) String titulo,
- @RequestParam int pagina,
- @RequestParam int qtd){

Paginação

- Agora é necessário criar um objeto **Pageable** via **PageRequest** e passar **pagina** e **qtd** como parâmetro.
- @GetMapping
- public **Page**<PostDto> listar(@RequestParam(required = false) String titulo, int pagina, int qtd){
-
- **Pageable pageable = PageRequest.of(pagina, qtd);**
-
- if((titulo!=null) && (!titulo.equals(""))) {
- return PostDto.converte(repository.findByTitulo(titulo,pageable));
- }
-
- return PostDto.converte(repository.findAll(pageable));
- }

Paginação

- Adicione **@EnableSpringDataWebSupport** acima do **método main**
- Assim podemos simplificar passando o **Pageable** como parâmetro do método listar.
- @GetMapping
- public **Page**<PostDto> listar(@RequestParam(required = false) String titulo, **Pageable pageable**){
-
- if((titulo!=null) && (!titulo.equals(""))) {
- return PostDto.converte(repository.findByTitulo(titulo,pageable));
- }
-
- return PostDto.converte(repository.findAll(pageable));
- }

Paginação

- A URL deve ter os parâmetros page e size (em inglês mesmo)
- http://localhost:8080/pessoas?**page**=0&**size**=3
- Podemos passar ainda um parâmetro de ordenação chamado **sort com o nome do campo para ordenar**
- http://localhost:8080/pessoas?page=0&size=3&sort=sobrenome,asc
-
-

Swagger

- Adicionar o Swagger a um projeto Spring Boot é uma excelente maneira de documentar e testar suas APIs REST
- O Swagger é integrado ao Spring Boot por meio do Springdoc OpenAPI.
- Primeiro Add a dependência:
- <dependency>
- <groupId>org.springdoc</groupId>
- <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
- <version>**2.6.0**</version>
- </dependency>
- Verifique a versão mais recente ↠ https://springdoc.org/

Swagger

- Agora vamos configurar. Crie uma classe de configuração no pacote .config.
- @Configuration
- public class SwaggerConfig {
-
- @Bean
- public OpenAPI customOpenAPI() {
- return new OpenAPI()
- .info(new Info()
- .title("API de Vendas de Selos")
- .version("1.0")
- .description("Documentação da API de Vendas de Selos"));
- }
- }

Swagger

- Agora podemos anotar o controlador com informações que aparecem na Documentação. Por exemplo:
- import io.swagger.v3.oas.annotations.Operation;
- import io.swagger.v3.oas.annotations.responses.ApiResponse;
-
- @RestController
- @RequestMapping("/api/selos")
- public class SeloController {
-
- @Operation(summary = "Listar selos", description = "Retorna todos os selos disponíveis")
- @ApiResponse(responseCode = "200", description = "Lista de selos")
- @GetMapping
- public List<Selo> listarSelos() {
- return seloService.listarTodos();
- }
- }

Swagger

- Crie uma configuração no application.properties :

  springdoc.api-docs.path=/api-docs

  springdoc.swagger-ui.path=/swagger-ui.html

- Para acessar: http://localhost:8080/swagger-ui/index.html

Exercício

1. Crie uma API restfull para servir de backend para uma agenda telefônica.
    a. Cada Contato da Agenda pode ter até 3 números de contato
    b. Cada número pode ser categorizado como PESSOAL, COMERCIAL e WHATSAPP, TELEGRAM e OUTRO.
    c. A agenda permite cadastrar um endereço e um e-mail por contato.
    d. Pelo menos um número e nome do contato são obrigatórios (Valide isso)
    e. Pagine sua agenda para retornar no mário 3 registros por página
    f. Normalize o banco e crie entidades separadas para Contato e Número

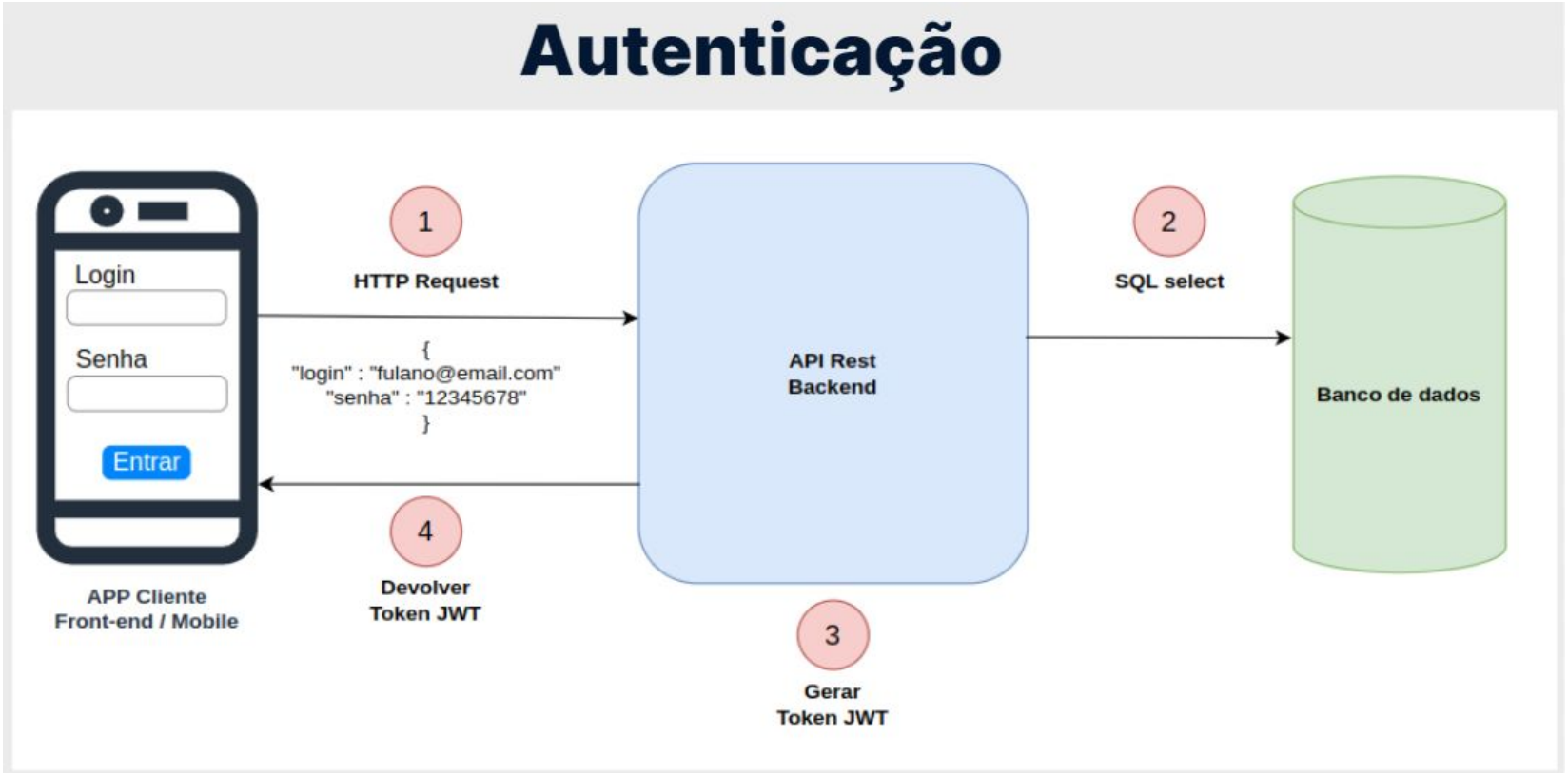
Segurança

- As nossas APIs estão abertas! Qualquer um com a URL pode listar, consultar, alterar e deletar!.
- No mundo real normalmente as APIs são protegidas com algum nível de segurança.
- O Spring tem um módulo focado em segurança chamado Spring Security.

Segurança

- O primeiro passo é add ele no pow.xml
-
- <dependency>
- <groupId>org.springframework.boot</groupId>
- <artifactId>**spring-boot-starter-security**</artifactId>
- </dependency>
-
-
- <dependency>
- <groupId>org.springframework.security</groupId>
- <artifactId>spring-security-test</artifactId>
- <scope>test</scope>
- </dependency>

Segurança


Autenticação
Login
Senha
Entrar
APP Cliente
Front-end / Mobile
1
HTTP Request
{
"login" : "fulano@email.com"
"senha" : "12345678"
}
4
Devolver
Token JWT
API Rest
Backend
3
Gerar
Token JWT
2
SQL select
Banco de dados

Segurança


Login + Senha
Token JWT/Refresh
Requisições (Token)
Recurso (200 OK)
Usuário
Servidor

Segurança

- Ao adicionar o Spring Security a aplicação passa a bloquear TODAS as requisições por padrão.
- Inicialmente vamos criar uma classe para guardar as configurações de segurança das nossas APIs chamada **SecurityConfigurations** em um pacote ".config.security"
- Essa classe deve ser anotada com **@EnableWebSecurity** e **@Configuration**
- Vamos criar um método que , INICIALMENTE, libera todos os bloqueios :
    - @Bean
- public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
- return http.csrf(csrf ↠ csrf.disable())
    - .sessionManagement(sess ↠ sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
- .build();
- }

Segurança

- Próximo passo é criar um Serviço de Autenticação no pacote de serviços
- Perceba que ele implementa **UserDetailsService (do Spring Security) e** assim deve implementar o método **loadUserByUsername**
- @Service
- public class **AutenticacaoService** implements **UserDetailsService** {
- @Override
- public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
- // TODO Auto-generated method stub
- return null;
- }
- }

Segurança

- Na sequência injetamos o Repositório de usuario e criamos lá o método

  UserDetails findByLogin(String username);

- @Service
- public class **AutenticacaoService** implements **UserDetailsService** {
- @Autowired
- private UsuarioRepository repository;
-
- @Override
- public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
- // TODO Auto-generated method stub
- return repository.findByLogin(username);
- }

Segurança

- Altere a entidade Usuario para que ela implemente UserDetails (e todos os métodos declarados)
- @Entity(name = "usuarios")
- @Setter
- @Getter
- @NoArgsConstructor
- @AllArgsConstructor
- public class Usuario **implements UserDetails**{
- .....
- @Override
- public Collection<? extends GrantedAuthority> getAuthorities() {
- return List.of(new SimpleGrantedAuthority("ROLE_USER"));
- }

Segurança

- Próximo passo é criar um AutenticacaoController para tratar as requisiçoes de login

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {
	@PostMapping
	public ResponseEntity efetuarLogin(@RequestBody  DadosAutenticacao dados) {
 ......
	}
}
```

Segurança

- Antes de seguir vamos criar um DTO só para o login
-
-
- public record DadosAutenticacao(String login, String senha) {
- }
-

Segurança

- O processo de autenticação está na classe **AutenticacaoService**. Precisamos chamar o método **loadUserByUsername**, já que é ele que usa o repository para efetuar o select no banco de dados.
- Porém, não chamamos a classe service de forma direta no Spring Security. Temos outra classe do Spring que chamaremos e é ela que vai chamar a AutenticacaoService.
- No controller, precisamos usar a classe **AuthenticationManager** do Spring, responsável por disparar o processo de autenticação.

Segurança

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

    @Autowired
    private AuthenticationManager manager;

    @PostMapping
    public ResponseEntity efetuarLogin(@RequestBody DadosAutenticacao dados) {
        var token = new UsernamePasswordAuthenticationToken(dados.login(), dados.senha())
        var authentication = manager.authenticate(token);

        return ResponseEntity.ok().build();
    }
}
```

Segurança

- A classe **AuthenticationManager** é do Spring. Porém, ele não injeta de forma automática o objeto AuthenticationManager, precisamos configurar isso no Spring Security. Como não configuramos, ele não cria o objeto AuthenticationManager e lança uma exceção.
- Assim é preciso voltar na classe **SecurityConfigurations** e incluir um método que permita a injeção da dependência.

Segurança

```
@Configuration
@EnableWebSecurity
public class SecurityConfigurations {

	@Bean
	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

		return http.csrf(ses -> ses.disable())
				.sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
					.build();

	}

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

Segurança

- Não é uma boa prática de segurança armazenar senhas no banco em texto livre
- Devemos usar algum algoritmo de hashing de senhas.
- Aqui vamos usar o BCrypt.
- Por exemplo
  - "SDFYJMJS"
  - Em BCrypt fica: $2a$12$moU2oz7EwhWhYUShDa6tgexwSNjgVJU3rZmWN.uubjYoImGYKSZAC

Segurança

- Para que o Spring entenda a codificação BCrypt é preciso modificar o config mais uma vez adicionando o método:
-

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

Segurança: Resumo até aqui

- A requisição de login chega na classe **AutenticacaoController**.java. Nela, criamos o método efetuarLogin, recebendo o DTO **DadosAutenticacao**.
- Usamos, além disso, as classes do Spring Security para disparar o processo de autenticação. O DTO do Spring Security é **UsernamePasswordAuthenticationToken**. Neles, passamos o login e a senha que chegam ao DTO
- Usamos, também, a classe AuthenticationManager, do Spring Security, para disparar o processo de autenticação.
- Agora o foco será criar e retornar um Token JWT como produto da autenticacao.

Segurança: JSON Web Token (JWT)

- JWT é um padrão da indústria (RFC 7519) que permite requisições seguras entre partes.
- É um padrão utilizado para a geração de tokens, que nada mais são do que Strings, representando, de maneira segura, informações que serão compartilhadas entre dois sistemas (https://jwt.io)
- No projeto vamos usar uma implementação do JWT chamada de AuthO:
- <dependency>
- <groupId>com.autho</groupId>
- <artifactId>java-jwt</artifactId>
- <version>4.4.0</version>
- </dependency>

Segurança: JSON Web Token (JWT)

- Para manter o padrão, vamos criar um serviço chamado JWTokenService e nele incluir a geração do Token.

```
@Service
public class JWTokenService {

    public String gerarToken(Usuario usuario) {
        try {
            var algoritmo = Algorithm.HMAC256("12345678");
            return JWT.create()
                .withIssuer("Aula de PWEB")
                .withSubject(usuario.getLogin())
                .withExpiresAt(dataExpiracao())
                .sign(algoritmo);
        } catch (JWTCreationException exception){
            throw new RuntimeException("erro ao gerrar token jwt", exception);
        }
    }

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

Segurança: JSON Web Token (JWT)

- A criação do token é feita no método **gerarToken**.
- A biblioteca JWT é usada na criação.
- Nela escolhemos o algoritmo de criação HMAC256 (existem MUITOS outros).
- Podemos incluir no token um conjunto de informaçoes
  - "Dono": **.withIssuer**("API de PWEB")
  - Usuario: **.withSubject**(usuario.getLogin())
  - Data de expiração: **.withExpiresAt**(dataExpiracao())
  - id do usuario: **.withClaim**("id", usuario.getId())
  - e muitas outras infos podem ser incluídas usando o **withClaim**
- Depois é só chamar isso no controller de autenticação

Segurança: JSON Web Token (JWT)

```
@RestController
@RequestMapping("/login")
public class AutenticacaoController {

	@Autowired
	private AuthenticationManager manager;

	@Autowired
	private JWTokenService tokenService;

	@PostMapping
	public ResponseEntity efetuarLogin(@RequestBody DadosAutenticacao dados) {
		var token = new UsernamePasswordAuthenticationToken(dados.login(), dados.senha());
		var autentication= manager.authenticate(token);

		return ResponseEntity.ok(tokenService.gerarToken((Usuario)autentication.getPrincipal()));
	}
}
```

Segurança: JSON Web Token (JWT)

- Agora o endpoit de login devolve um TOKEN JWT: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJBUEkgVm9sbC5tZWQiLCJzdWIiOiJtYWxlbmFAZW1haWwuY29tIiwiZXhwIjoxNzExNTY3ODA2fQ.VvEeK9xTrFXIuaD75pFFTxoWC-ENFNcgFt9HgDbgH-A
- Acesse https://jwt.io/ e faça o decode desse token apenas para testar.
- Depois disso vamos organizar o retorno do token.
- Ao invés de retornar uma String, vamos retornar um record **DadosTokenJWT com apenas um atributo token**

Segurança: JSON Web Token (JWT)

```
@PostMapping
    public ResponseEntity efetuarLogin(@RequestBody DadosAutenticacao dados) {
    var authenticationToken = new UsernamePasswordAuthenticationToken(dados.login(), dados.senha());
    var authentication = manager.authenticate(authenticationToken);

    var tokenJWT = tokenService.gerarToken((Usuario) authentication.getPrincipal());

    return ResponseEntity.ok(new DadosTokenJWT(tokenJWT));
  }
```

Segurança: JSON Web Token (JWT)

- o retorno agora é um JSON:
- {

  "token":
  "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJQVoVCIEFQSSIsInN1YiI
  6Im1hbGVuYUBlbWFpbC5jb2oiLCJleHAiOjE3MTE1Njg5MDV9.QFK6OYPVNsB
  apOSZBSSB2IRCp2iE-hRe_6gPKd3CWcc"

  }

Segurança: JSON Web Token (JWT)

- Agora voltaremos à classe "**JWTokenService**.java" na IDE. Nela, precisamos passar uma senha secreta na linha de criação do algoritmo, o que é indispensável para fazer a assinatura do token.
-
- Nas aulas anteriores, havíamos passado "12345678" como senha. Como passar a senha em texto dentro do código não é uma boa prática de segurança, vamos fazer a leitura dessa senha de algum lugar.
- O primeiro passo será remover o "12345678" do código. No lugar dela, passaremos um atributo chamando secret. Vamos declarar o atributo dentro da classe **JWTokenService**, com a linha de código private String secret;

Segurança: JSON Web Token (JWT)

- Na linha acima de private String secret;, passaremos a anotação @Value
- Cuidado ao importar! Há o Value do Lombok e o value do Spring Framework. O que nos interessa é o segundo.
- Entre aspas, como parâmetro, passaremos "${api.security.token.secret}"
- Lá no application.properties escolhemos a senha....essa senha é importante para garantir que ninguém gere um token JWT "por fora" do sistema.
- Depois de tudo isso, Autenticar, Gerar e Devolver um TOKEN JWT, vamos seguir com a fase de AUTORIZAÇÃO: o que é ou não é permitido para um usuário autenticado

Segurança: Autorização

- Para criar a lógica de autorização precisamos receber um token, validá-lo e aí decidir se uma requisição pode ou não ser executada.
- Não vamos fazer isso repetidas vezes em cada método de cada controller. Vamos usar um **Filter**
- **Filter** é um dos recursos que fazem parte da especificação de Servlets, a qual padroniza o tratamento de requisições e respostas em aplicações Web no Java. Ou seja, tal recurso não é específico do Spring, podendo assim ser utilizado em qualquer aplicação Java.
- É um recurso muito útil para isolar códigos de infraestrutura da aplicação, como, por exemplo, segurança, logs e auditoria, para que tais códigos não sejam duplicados e misturados aos códigos relacionados às regras de negócio da aplicação

Segurança: Autorização

- Para criar um Filter, basta criar uma classe e implementar nela a interface Filter (pacote jakarta.servlet). Por exemplo:
- @**WebFilter**(urlPatterns = "/api/**")
- public class **LogFilter** implements Filter {
-
- @Override
- public void **doFilter**(**ServletRequest** servletRequest, **ServletResponse** servletResponse, **FilterChain** filterChain) throws IOException, ServletException {
- System.out.println("Requisição recebida em: " + LocalDateTime.now());
- filterChain.doFilter(servletRequest, servletResponse);
- }
-
- }

Segurança: Autorização

- O método doFilter é chamado pelo servidor automaticamente, sempre que esse filter tiver que ser executado, e a chamada ao método filterChain.doFilter indica que os próximos filters, caso existam outros, podem ser executados.
- A anotação @WebFilter, adicionada na classe, indica ao servidor em quais requisições esse filter deve ser chamado, baseando-se na URL da requisição.
- Aqui, utilizaremos outra maneira de implementar um filter, usando recursos do Spring que facilitam sua implementação.

Segurança: Autorização

- Vamos criar um filtro no projeto, para interceptar requisições. O que queremos é fazer a validação do token antes que ele caia no controller.
- O nome da classe será "SecurityFilter" no pacote config
- Como o Spring não conseguirá carregar a classe automaticamente no projeto, precisaremos passar a anotação @Component no código.
- A classe deve herdar de OncePerRequestFilter (que é um filter do Spring) e vamos implementar o método doFilterInternal.
- Para testar imprima um System.out.println("Chamou o filtro")

Segurança: Autorização

```
@Component
public class SecurityFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        System.out.println("FILTRO CHAMADO");
        filterChain.doFilter(request, response);
    }
}
```

Segurança: Autorização

- O método filterChain.doFilter(request, response), garante que fluxo da requisição siga. Se ele não for chamado a requisição para no filtro!
- O próximo passo é agora receber um token JWT e validar
- O token é enviado por um cliente no cabeçalho de uma requisição HTTP. o campo se chama **Authorization**

Segurança: Autorização

- O método filterChain.doFilter(request, response), garante que fluxo da requisição siga. Se ele não for chamado a requisição para no filtro!
- O próximo passo é agora receber um token JWT e validar
- O token é enviado por um cliente no cabeçalho de uma requisição HTTP. o campo se chama **Authorization**
- @Override
- protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
- throws ServletException, IOException {
- var token = request.**getHeader**("Authorization");
- if (token == null || token.isEmpty() || !token.startsWith("Bearer ")) {
- response.setStatus(401);
- }else{
- token = token.replace("Bearer ", "");
- }
- filterChain.doFilter(request, response);
-
- }

Segurança: Autorização

- Agora que temos o token recuperado é preciso validar. Para isso, vamos incluir um método de validação na classe **JWTokenService**

```
public String getSubject(String tokenJWT) {
    try {
        var algoritmo = Algorithm.HMAC256(secret);
        return JWT.require(algoritmo)
                .withIssuer("Aula de PWEB")
                .build()
                .verify(tokenJWT)
                .getSubject();
    } catch (JWTVerificationException exception) {
        throw new RuntimeException("Token JWT inválido ou expirado!");
    }
}
```

Segurança: Autorização

```
public String recuperarToken(HttpServletRequest request) {
        var token = request.getHeader("Authorization");
        if (token == null || token.isEmpty() || !token.startsWith("Bearer ")) {
            return null;
        }
        return token.replace("Bearer ", "");
    }
```

Segurança: Autorização

- Em seguida injetamos **JWTokenService** no SecurityFilter e na sequência validamos o token no **doFilter**.

```
@Autowired
private JWTokenService tokenService;
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
                throws ServletException, IOException {
        var token = recuperarToken(request);
        if(token!=null) {
            var login = tokenService.getSubject(token);
            System.out.println("Login: " + login);
        }
        filterChain.doFilter(request, response);
```

Segurança: Autorização

- Agora que recuperamos e validamos o token é preciso "autenticar" as requisições com o token válido.
- Precisamos voltar no SecurityConfigurations e alterar o método securityFilterChain para liberar a requisição de login (Claro!!) e so liberar as demais se elas estiverem autenticadas.

Segurança: Autorização

```
@Autowired
        private SecurityFilter securityFilter;
@Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
                return http.csrf(csrf -> csrf.disable())
                  .sessionManagement(sm ->
sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                  .authorizeHttpRequests(req -> {
                     req.requestMatchers(HttpMethod.POST, "/login").permitAll();
                     req.anyRequest().authenticated();
                  })
                  .addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
                  .build();
        }
```

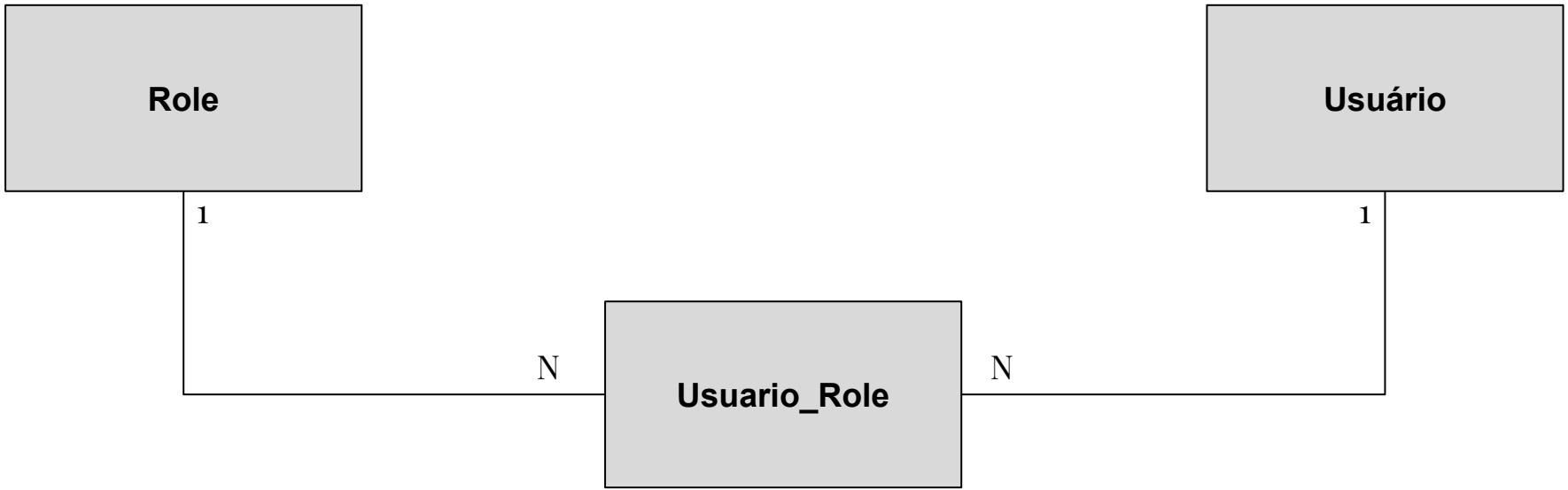
Segurança: Autorização

- Por fim é preciso autorizar as requisições quando o token for válido

Segurança: Autorização

```
@Autowired
	private JWTokenService tokenService;

	@Autowired
	UsuarioRepository usuarioRepository;

	@Override
	protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
			throws ServletException, IOException {
		var token = recuperarToken(request);
		System.out.println("Token: " + token);
		if(token!=null) {
			var login = tokenService.getSubject(token);
			var usuario = usuarioRepository.findByLogin(login);
			var authentication = new UsernamePasswordAuthenticationToken(usuario, null, usuario.getAuthorities());
			SecurityContextHolder.getContext().setAuthentication(authentication);

		}
		filterChain.doFilter(request, response);

	}
```

E Se quisermos incluir Perfis de Usuários?(Lembre de "Zerar o BD")



Incluindo Perfil: Criar a entidade "Role" e o seu RoleDto

```
@Entity(name = "roles")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Role implements  GrantedAuthority {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String role;

	public Role(RoleDto roleDto) {
		this.id = roleDto.id();
		this.role = roleDto.role();
	}

	@Override
	public String getAuthority() {
		// TODO Auto-generated method stub
		return role;
	}

}
```

Incluir o atributos Roles na entidade usuários e criar a relação NxN

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String login;
    private String senha;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
      name = "usuarios_roles",
      joinColumns = @JoinColumn(name = "usuarios_id"),
      inverseJoinColumns = @JoinColumn(name = "roles_id")
      )
    private List<Role> roles= new ArrayList<Role>();
```

Carga Inicial

- INSERT INTO usuarios (login, nome, senha) VALUES(nextval('malena@gmail.com', 'Maria Helena, '$2a$12$ut.cwRAquAgEhTseyFydReAzHBcMaa7uc3wZDFABUKG19H9Z8MQ.i'); (Senha 123456 em BCrypt)
- INSERT INTO roles (role) VALUES('ROLE_USER');
- INSERT INTO roles (role) VALUES('ROLE_ADMIN');
- INSERT INTO usuarios_roles (usuarios_id, roles_id) VALUES(1, 1);
- INSERT INTO usuarios_roles (usuarios_id, roles_id) VALUES(1, 2);

Alterar o construtor de Usuario

```
    public Usuario(UsuarioDto usuarioDto) {
        this.nome = usuarioDto.nome();
        this.login = usuarioDto.login();
        this.senha = usuarioDto.senha();
        this.roles = usuarioDto.roles().stream().map(Role::new).toList();
    }
```

Alterar o método get atributos getAuthorities()

```
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // Passa a retornar a lista de "roles"
        //return List.of(new SimpleGrantedAuthority("ROLE_USER"));
        return roles;
    }
```

Incluir anotação em @EnableMethodSecurity

- @Configuration
- @EnableWebSecurity
- @**EnableMethodSecurity**(securedEnabled = true)
- public class **SecurityConfigurations** {
- .....
-

Incluir anotação em @Secured("ROLE_XXX") no controlador

- POR EXEMPLO:
- @DeleteMapping("/{id}")
- 	@Transactional
- 	@**Secured("ROLE_ADMIN")**
- 	public ResponseEntity<UsuarioDto> deletar(@PathVariable Long id) {
-
- 		return usuarioService.deletar(id);
- 	}
-