# Treinamento #1 [GIT e Github](https://www.udemy.com/course/curso-de-git-e-github-essencial/learn/lecture/17973515#questions)

repositório BARE: são repositório sincronizáveis, para iniciar um localmente usa-se

git init –bare

git fetch // baixa as atualizações da branch sem alterar os arquivos locais

git diff main origin/main // verifica a diferença entre as branchs atual e espelhada local.

git fetch origin pull/<númeroDoPullRequest>/head:<nomeDaNovaBranch>

// baixa uma requisição de pull por parte de um colaborador e transforma em uma branch.

git tag: cria pontos de versões operacionais, onde não se pode realizar novos pushs para a mesma, ainda assim pode-se clonar tais tags para uma branch.

git clone/ cria um repositório automaticamente vinculado a um endereço.

git checkout -- . / volta arquivos modificados (não adicionados/commitados) à forma do commit mais recente.

git checkout -- <nomeArquivo> / mesma coisa do anterior, mas com arquivo específico.

git checkout HEAD -- . / volta arquivos modificados ao estágio anterior/ponto do commit

git checkout HEAD -- <nomeArquivo> / volta arquivo modificado em específico para ponto do commit.

git reset HEAD^ --hard / volta para o penúltimo commit, tornando-o o último commit

git revert <commit id> / desfaz alterações realizadas pelo commit informado

git commit –amend –m “<mensagem>” faz commit em cima de outro commit.

git remote set-url origin https://dansousac2@github.com/dansousac2/dansousac2.github.io.git

// configurar usuário padrão para as operações de git.

= = = = = = = = = = = = = = = = = = = = = = = = = = =

Nesse Treinamento preparatório também foi trabalhado com manipoulação de arquivos MarkDown (.md).

# Treinamento #2 [TDD e Java: testes automatizados com JUnit](https://cursos.alura.com.br/course/tdd-java-testes-automatizados-junit/task/88553)

JUnit – biblioteca do Java que facilita testar o código desenvolvido, colocando os casos em que o mesmo deve mostrar POSITIVO para implementação bem sucedida.

TDD – Test Driven Development (Desenvolvimento Direcionado/Orientado à Testes)

Testes->Desenvolvimento->REFATORAÇÃO->testes

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

Adicionando JUnit às dependências

Versão atualizada no site: <https://search.maven.org/search?q=g:org.junit.jupiter>

Versão 5.8.2//

colocar no arquivo **pom.xml** //

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.7.0</version>

<configuration>

<release>11</release>

</configuration>

</plugin>

</plugins>

</build>

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

Notações JUnit//

**@Test** // diz que o método e para testes do JUnit

**@BeforeEach** // diz que o método deve ser inicializado antes daquesles que são do tipo @Test

**@BeforeAll** (deve ser estático) // diz que o método será executado antes dos métodos @Test

**@AfterEach** // diz que o método deve ser chamado após cada método @Teste ser executado

**@AfterAll** // diz que o método deve ser chamado após a conclusão dos métodos @Test

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

🡪Quando um método deve nos retornar um valor exato, utilizamos um tipo com valor específico (esperado) e um valor ou variável que obtivemos: *assertEquals*(new BigDecimal("250.00"),bonus);

🡪Quando um método deve lançar exceção, usamos com argumentos nesse método abaixo o tipo de exceção e o método que a provoca (como o método não é chamado diretamente, usa-se **lâmbida**: APENAS ISSO BASTA/ não incluir a chamada direta do objeto com o método que retorna erro.

*assertThrows*(IllegalArgumentException.class,

() -> bs.calcularBonus(new Funcionario("Fulano", LocalDate.*now*(), new BigDecimal(11000))));

} // "() ->" = lâmbida.

Nesse último caso também **podemos** tratar com try catch no teste para visualizar detalhes, como a mensagem do erro.

# Treinamento #3 [Mocks em Java: Conhecendo o Mockito](https://cursos.alura.com.br/course/mocks-java-mockito)

Mocks são estruturas que simulam o comportamento de alguma classe definida. Servem principalmente para quando queremos testar um método sem nos preocuparmos com suas dependências entre classes.

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

Dependências//

<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.3.0</version>

<scope>test</scope>

</dependency>

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

* criando mock por método estático//

tipo nome = Mockito.mock(tipo.class);

* Criando mock por anotação//

🡪Colocando marcação do atributo que será um mock

@Mock

private LeilaoDao leilao;

🡪Para que o o Mockito crie os mocks (essa parte costuma ser usada com @BeforeEach//

MockitoAnnotations.openMocks(this);

🡪Para que o Mockito devolva alguma informação em um determinado método//

Mockito.when(mock.met).thenReturn(value);

🡪Lançando exceção em método que possui qualquer parâmetro de entrada//

Mockito.when(mock.met(**Mockito.any()**)).thenTrow(Exception);

Obs: tratar isso com **Try/Catch**, pois ao lançar exceção o JUnit para o teste.

🡪Verificar se um método do mock está sendo executado//

Mockito.verify(mock).métodoASerAnalisado;

🡪Verificar se não houve interações com o mock//

Mockito.verifyNoInteractions(mock);

🡪Verificar um objeto que foi criado dentro da classe de teste//

@Captor

private ArgumentCaptor<Folha> capFolha;

E no método mockado onde o objeto é usado, ex: listar(Folha), usamos listar(capFolha.capture());

🡪Podemos utilizar esse objeto capturado da seguinte maneira//

Folha folha = capFolha.getValue();

🡪<[Dicas para simular data com Mokito](https://gist.github.com/dansousac2/b71e7f7f40a1ed1f67de2ee9e4e84217)>

# Treinamento #4 [SOLID com Java: Princípios da programação orientada a objetos](https://cursos.alura.com.br/course/solid-orientacao-objetos-java/task/85731)

|  |  |  |
| --- | --- | --- |
| **Principle/Princípio** | **Tradução** | **Objetivo** |
| **S**ingle Responsibility | Responsabilidade Única | Indica que cada classe deve conter apenas sua abstração essencial, ser responsável por funções inerentes apenas a esta |
| **O**pen Closed | Aberto Fechado | Entidades de software (classes, funções, módulos) devem estar abertas para extensões e fechadas para modificações.  Significa que devemos poder criar novas funcionalidades e estender o sistema sem precisar modificar muitas classes já existentes. |
| **L**iskov Substitution | Substituição de Liskov | Devemos poder substituir classes base por suas classes derivadas em qualquer lugar, sem problema.  Se uma herança compromete a coesão de uma subclasse, alternativamente devemos usar outra lógica (composição por exemplo) como substituta a herança. |
| **I**nterface Segregation | Segregação de Interface | Uma classe não deve ser forçada a depender de métodos que não utilizará.  Se há algum método divergente em uma interface para uma classe que a implementa, cria-se uma nova interface com essa “diferença”, e esta interface só será implementada por classes que lidam com essa “diferença”. |
| **D**ependency Inversion | Inversão de Dependência | Abstrações não dependem de implementações, são as implementações que dependem das abstrações.  Ex: ValidacaoPercentual (implem.) / ValidacaoPeriodicidade (implem.) / ValidacaoReajuste (abstração) |

* São boas práticas de programação orientada à objetos.
* Deixa o código mais fácil de manter e estender.

# Treinamento #5 [Design Patterns em Java I: Introdução às boas práticas de programação](https://cursos.alura.com.br/course/introducao-design-patterns-java)

* Padrão de projeto: solução comum à um problema conhecido.

Padrões podem ser:

* Criacionais
* Estruturais
* Comportamentais
* COMPORTAMENTAIS (alguns deles)
* Strategy
* Chain of Responsability
* Template Method
* State
* Command
* Observer

## 🡺Strategy

O *Strategy* resolve o problema da existência de diversos algoritmos para uma ação, resultando na possibilidade de vários ifs. Dessa forma, trocam-se as condicionais por **classes e Interface**. **Sabemos** o tipo de lógica (classe) a ser aplicada.

No lugar de tantos if’s utilizamos as possibilidades como classes concretas, e também podemos usar polimorfismo, por meio de interfaces por exemplo, para garantir mais coesão nas relações intraclasse.

## 🡺Chain of Responsibility

O *Chain of Responsibility* resolve o problema da existência de diversos algoritmos para uma ação, resultando na possibilidade de vários ifs. Dessa forma, trocam-se as condicionais por **classes e herança**, onde a SUPER (abstrata) tem um atributo de seu próprio tipo chamado “próximo”. **Não sabemos** o tipo de lógica (classe) a ser aplicada.

Não sabemos qual a estratégia/regra que será aplicada (classe contendo a regra, sendo que essas classes são de um mesmo tipo, “Desconto” por exemplo, temos vários tipos). Usamos então, uma SUPER contendo um atributo de seu próprio tipo, e quando instanciamos um primeiro objeto desse tipo, o próximo é passado via construtor, juntamente com o próximo do próximo:

objA(objB(objC));

Cada subclasse implenta um método abstrato da SUPER implementando sua lógica e devolvendo um valor OU chamado a próxima classe (atributo da SUPER).

## 🡺Template Method

O *Template Method* resolve o problema da existência de parte de código duplicado em métodos. Como solução, é delegada à uma SUPER, parte da execução do código repetido, cabendo às subclasses realizarem coisas como verificações e a sua própria implementação do método em questão.

## 🡺State

O *State* resolve o problema de um objeto se comportando de maneiras diferentes dependendo de qual seja seu estado. Como solução, cada estado passa a ser uma classe específica de uma SUPER abstrata.

## 🡺Command

O *Command*, como também o *Command* Handler, resolvem o problema onde temos que executar algumas ações (salvar em banco, enviar e-mails, etc.) onde essas funções ficam fixas em parte do código. Como solução, temos que cada caso de uso deve ser extraído para uma classe específica, ao invés de estar no arquivo da CLI, controller ou algo do tipo. Pode-se utilizar o *Command Handler* para receber as “classes ações”, enquanto que a classe *Command* recebe os dados a serem passados para o *handler*.

## 🡺Observer

O padrão *Observer(Observador)* ou *Listener(Ouvinte)*, resolve o problema onde temos quedisparar uma lista crescente de ações quando um certo método for chamado. Como solução, temos que uma lista é criada com essas ações, e o método que dispara todas essas ações recebe essa lista, mandando cada item executar sua ação. Os itens da lista são os observadores/ouvintes que esperam ser chamados para executar sua ação.

# Treinamento #6 (A) Spring Boot Essentials

Framework criado com intuito stand alone, sem dependências e com inicialização rápida. Cria aplicações com o mínimo de esforço necessário.

Solução/ problema de porta em uso:

Em algum terminal:

netstat -ona | findstr :8080 | findstr LISTENING // lista processo na porta selecionada (8080)

taskkill /F /PID <pid> // parando processo de ID especificado

# Treinamento #6 (B) // [Spring Boot API REST: Construa uma API](https://cursos.alura.com.br/course/spring-boot-api-rest) [ALURA]

## 🡺Criando o projeto: [start.spring.io](https://start.spring.io/)

## 🡺Observações

### Postman

|  |  |
| --- | --- |
| **Comando** | **Obs** |
| GET (busca) | Segue apenas o mapeamento HTML informado no código |
| POST (criar) | Necessita de um corpo Jsom (Body + row) com os campos/atributos a serem atualizados, e de um cabeçalho dizendo o tipo de informação que será enviado (KEY = Content-Type; VALUE = application/json) |
| PUT (atualiza) | Idem acima. |
| DELETE (exclui) | Só precisa informar de acordo com o código, que pode ser o ID do objeto |

1. O Jackson é usado pelo Spring para fazer a converlçao para JaSom e assim carregar na tela os dados String.
2. Não é uma boa prática retornar entidades JPA nos métodos dos *controllers*, sendo mais indicado retornar classes que seguem o padrão **DTO** (*Data Transfer Object*), que passa apenas os dados selecionados, funcionando como um tipo de filtro (exemplo: Pessoa e PessoaDto).
3. Os principais conceitos sobre o modelo arquitetural REST: **recursos, URIs, verbos HTTP, Representações e comunicação stateless.**
4. Não foi utilizado o padrão DAO para o banco de dados, mas sim o REPOSITORY, que usa uma interface que implementa uma outra interface do SpringData que possui alguns métodos prontos (CRUD).
5. Para utilizar o JPA no projeto, devemos incluir o módulo Spring Boot Data JPA, que utiliza o Hibernate, por padrão, como sua implementação.
6. Para configurar o banco de dados da aplicação, devemos adicionar as propriedades do datasource e do JPA no arquivo src/main/resources/application.properties.
7. Para acessar a página de gerenciamento do banco de dados H2, devemos configurar o console do H2 com propriedades no arquivo src/main/resources/application.properties;
8. Para que o Spring Boot popule automaticamente o banco de dados da aplicação, devemos criar o arquivo src/main/resources/data.sql contendo os comandos SQL.
9. Para criar um Repository, devemos criar uma interface, que herda da interface JpaRepository do Spring Data JPA;
10. Para criar consultas que filtram por atributos da entidade, devemos seguir o padrão de nomenclatura de métodos do Spring, como por exemplo findByCursoNome; (aqui estamos acessando Curso e dentro dele pegando Nome, o que é o mesmo que findByCurso\_Nome; para o String).
11. Para criar manualmente a consulta com JPQL, devemos utilizar a anotação @Query, inserindo manualmente a consulta.
12. Dto 🡪 dados saem da API para o cliente, Form (tipo de Dto, padrão de nomenclatura) 🡪 dados que chegam do cliente para a API.
13. Cod 200 🡪 a requisição foi processada com sucesso, código genérico. O código 201 diz que o processo foi concluído com êxito e que um novo recurso foi criado no servidor. Este último usa ResponseEntity<tipo> (ver nas notações do Spring).
14. Bean Validation 🡪 API do Java que se integra com o SpringBoot para validação de campos (ver anotação @Valid)
    1. Quando um erro é lançado usando o Bean Validation, o Spring lança uma MethodArgumentNotValidException. Esta classe possui um método que gera uma List de campos de erros// List<FieldError>

exception.getBindingResult().getFieldErrors();

1. A partir da versão 2.3.0 do Spring Boot o Bean Validation não vem mais incluído automaticamente no projeto, sendo necessário adicionar essa dependência no arquivo pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

1. A partir da versão 2.5 do Spring Boot o método getOne(), das interfaces repository, foi marcado como deprecated, sendo que agora devemos utilizar o novo método getById().
   1. Ex:

//metodo getOne antigo:

//Topico topico = topicoRepository.getOne(id);

**//Utilizar agora o metodo getById:**

Topico topico = topicoRepository.getById(id);

## 🡺Anotações

|  |  |
| --- | --- |
| **Anotação** | **Função** |
| @Controller  public class \*\*\* | O Spring encontra a classe e faz seu gerenciamento. |
| @RequestMapping(“/”)  public String hello() | O método deve ser chamado no endereço localhost:8080/ |
| @ResponseBody  public Sting hello() {  return “Hello Wolrd”;  } | Faz com que o retorno seja uma String e não umapágina de nome “Hello Wolrd”. |
| @RestController  puvlic class \*\*\* | Assume que todo método da classe é do tipo @ResponseBody |
| @Autowired | Injeção de dependências |
| @GetMapping  @PostMapping | Esses dois são utilizados para fazer um diferenciamento nos métodos de uma mesma classe, à exemplo, quando usamos @RequestMapping(“/topicos”) a nível de classe, utilizamos as notações ao lado para realizar distinções nos métodos dessa classe.  enquanto o Get devolve algo dentro de /topicos, o Post altera algo em /topicoslj |
| (@RequestBody classe nome) | Informa que o objeto é para ser pego no corpo da requisição e não na URL informada. É usado com @PostMapping  “Indicar ao Spring que os parâmetros enviados no corpo da requisição devem ser atribuídos ao parâmetro do método” |
| Retorno/Classe do Spring  ResponseEntity<tipo> | É uma classe do Spring que devolve um objeto no corpo da resposta, podendo ser um código de URL, como o 200 (ok) ou o 404 (URL inválida).  Possui métodos estáticos.  O tipo não pode ser do domínio da aplicação, usa-se uma classe Dto como devolutiva da API para o cliente.  Para retornar esse objeto precisa-se de uma Uri e um corpo (body). |
| Classe do Spring  UriComponentsBuilder **nome**  Possui o método estático seguinte para endereçamento dinâmico, que devolve um objeto nativo do Java, o **Uri**.  **nome**  .path(“/tópico/{**id**}”)  .buildAndExpand(**topico.getID()**)  .toUri();  Na mesma linha, partindo de “nome”. | Deixa preparado para usar uma url quando o site for implementado. Passando apenas o caminho do recurso e o atributo de identificação (key). |
| Método(@Valid objeto nome) | Integralção de API do Java com Spring. O objeto deverá ser verificado, com as devidas anotações feitas neste objeto. |
| @RestControllerAdvice  Public class \*\*\*  - Por padrão devolve status de cod. 200 ao cliente. Isso pode ser alterado com a anotação @ResponseStatus | Quando há validações (@Valid) usa-se esta anotação para APIs Rest para fazer o tratamento de exceções, de modo a deixar mais claro o motivo do erro ao usuário final. |
| @ResponseStatus  (code = HttpStatus  .BAD\_REQUEST)  (Linha única) | Diz que o tratamento do erro não deve devolver 200 (ok) mas sim status de erro de requisição. |
| Class do Spring// MessageSource | Pega a mensagem de erro de acordo com idioma configurado.  String mensagem =  messageSource.getMessage(e, LocaleContextHolder.*getLocale*());  🡪descobrir qual o Locale atual. |
| Metodo  (@PathVariable classe nome)  Ex:  *@GetMapping*("/{id}")  public TopicoDto detalhar(*@PathVariable* Long id) {  Também pode usar outro nome no lugar do “id”, colocando a anotação no @PathVariable(“id”) classe nome | Avisa ao Spring que a variável esperada virá no corpo da URL, não utilizando “?”, mas sim “/” na URL. |
| @PutMapping  Public obj metodo()  @PetMapping  Public obj metodo() | - Sobrescrever o recurso inteiro  - Modifica um ou apenas lguns campos  Na aplicação não há como saber, a menos que seja verificado. O @PutMapping então passa a ser usada com mais frequência. |
| @Transactional  Metodo() | Avisa ao Spring para commitar ao final da execução do método, as alterações realizadas no BD. |
| @DeleteMapping  Public metodo() | Usando lógica de exclusão no Spring. |
| Método da JpaRepository (interface)  findById(id) | Usado no lugar do método “getOne” (depreciado em versões mais novas do SpringBoot) (que lança exceção no corpo na mensagem). |

# Treinamento #5b // [Spring Boot API Rest: Segurança da API, Cache e Monitoramento](https://cursos.alura.com.br/course/spring-boot-seguranca-cache-monitoramento)

## 🡺Paginação e ordenação

### 🡪Detalhes:

1. O REST (Representation State Transfer) é uma API que usa restrições HTTP para extrair, inserir, postar e deletar dados.
2. O Spring Data tem uma interface para paginação. Pageable paginacao = PageRequest.*of*(pagina, qtd); A vantagem de termos uma página como retorno é que podemos ver outras informações inportantes quanto aos elementos da página, como itens, quantas páginas há no total e itens no total, etc.
3. Utilizando Pageable a partir dos parâmetros da URL: método([...], Pageable paginação). Deve-se habilitar na classe **MAIN** o usu de suporte Web 🡪 **@EnableSpringDataWebSupport**. Partindo disto, os parâmetros dessa paginação na URL devem ser: **page** – **size** (número de elementos por página) – **sort(id,método(ASC,DESC))** (ordenação) [sort pode aparecer múltiplas vezes] 🡺 vert @PageableDefault

## 🡺 Utilizando memória em cache:

Uso em métodos que cujo retorno QUASE NUNCA É ATUALIZADO

1-add dependência, 2-add provedor, 3-habilitar na aplicação

1. Dependência

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

1. Deve também declarar um provedor de memória cache. Caso contrário o Spring usará um provedor que utiliza hash mapping (não é usado em aplicações comerciais).
2. Para habilitar na aplicação: @EnableCaching na class MAIN
3. Habilitar Hibernate para imprimir no console quando acessar o BD: Nas propriedades da JPA (application.properties), adicionar a seguinte linha:

spring.jpa.properties.hibernate.show\_sql=true.

Por padrão ele imprime todos os selects em uma única linha. Para organizar melhor, usamos: spring.jpa.properties.hibernate.format\_sql=true

## 🡺 Proteção com Spring Security

### Preparando o ambiente

1- dependência, 2- criar classe com as anotações de segurança, 3- habilitar esta classe (@EnableWebSecurity e @Configuration), 4- extender a classe HebSecurityConfigurerAdapter

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

🡺Após realizar o procedimento acima, o Spring bloqueia completamente o acesso a aplicação.

### Configurando

- Devemos sobrescrever os 3 métodos “configure” presentes na SUPER//

1º - configure(HttpSecurity http): Aqui devemos dizer quais endereços e tipos de métodos autorizados, como mostrado abaixo:

http.authorizeRequests()

.antMatchers(*HttpMethod*.***GET***, "/topicos").permitAll()

.antMatchers(*HttpMethod*.***GET***, "/topicos/\*").permitAll();

- depois disto, colocamos .anyRequest().authenticated() para dizer que as demais solicitações devem ser autenticadas. Devemos agora criar essa parte de autenticação.

1.2 - implements UserDetails: A classe que servirá para realizar as autenticações (User, por exemplo) deve implemetnar essa interface do Spring, e realizar também as anotações de persistência da JPA. Ao implementar os métodos, aqueles de retorno boleano devem ficar com TRUE se não fizerem parte da lógica da aplicação.

No método Collection<? extends GrantedAuthority> getAuthorities(), devemos retornar uma coleção de “Perfis” de usuário, que possuem uma ID (Long) e um NOME (String). Por sua vez, essa clase Perfis deve implementar de **GrantedAuthority**, para se adequar aos parâmetros do método em User, havendo obrigação de implementar apenas 1 método, o getAuthority() que devolve o atributo “nomePerfil” (String) de Perfis.

1.3 - .and().formLogin(): completando o método autorizeRequests do obj HTTP, temos esse método que faz com que outros elementos automaticamente sejam inseridos na lógica de autorização. Agora, quando não dor detectado usuário logado, por exemplo, o Spring cria uma interface automática que permite ao usuário fazer login.

2º - configure(AuthenticationManagerBuilder auth): indicar a classe que possui a lógica de autenticação.

2.2 - auth.userDetailsService(<classeService>): passamos a classe que contém a lógica de autenticação (se ela não existe, a criamos). Ela chega na classe de configuração através de injeção de dependências (@Autowired).

2.3 – a classe informada acima, deve implementar UserDetailsService (parecido com a da classe “Usuário” – UserDetails), e será um @Service do Spring. O método da SUPER exigirá aceso ao BD de usuários (o que pode ser feito criando uma interface repository de usuários – seguindo o padrão repository).

2.4 - <2.2>.passwordEncoder(new BCryptPasswordEncoder()): na mesma linha de comando do tópico 2.2, adicionamos este método para escolher a forma de criptografia desejada para a senha (depois desta modificação, a senha criptografada deve ser informada em data.sql, no lugar da senha não criptografada.

### Gerando Token com JWT

- No modelo REST, a boa prática nos diz que o ideal é que nossa aplicação seja stateless, ou seja, o servidor não mantém nada em sessão (não deveria guardar estados de informação de autenticação).

- Em aplicações web tradicionais o comum é se usar autenticação via Session. Esses dados são guardados no servidor, que utiliza coockies da máquina local para fazer a verificação de usuário. Não é usual fazer dessa forma.

- Com o Spring Security fazemos com que não sejam criadas sessões nos momentos que usuários sejam logados, onde o cliente manda as informações, via token.

1º - DEPENDÊNCIA//

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.1</version>

</dependency>

2º - .sessionManagement().sessionCreationPolicy(*SessionCreationPolicy*.***STATELESS***):

Na classe de segurança, no método configure(HttpSecurity http), Substituimos a linha .and().[...] por .and().csrf().disable() – aqui desabilitamos a securança para csrf, que é um tipo de ataque hacker que utiliza o sistema Session (que aqui não será usado).

Adicionar abaixo a linha

.sessionManagement().sessionCreationPolicy(*SessionCreationPolicy*.***STATELESS***), que avisa ao Spring Security que não é para criar sessões, utilizando assim o modo Stateless.

A partir de agora, não temos um formulário de login automático, já que apagamos a linha .adn()... que havia antes (seguindo os passos desse trabalho). Também perdemos o controller do Spring que fazia a lógica de verificação, devemos então criá-lo.

3º - Criar classe de autenticacaoController com devido método//

Método: public ResponseEntity<?> autenticar(*@RequestBody* *@Valid* LoginForm form)

Anotações presentes nessa classe criada: @RestController @RequestMapping(“...”) e @PostMapping no método.

Como é um controller, pelo padrão Dto, devemos criar um form de login (duas Strings: senha e usuário), que servirá como parâmetro do nosso método que verifica a autenticação.

4º - .antMatchers(*HttpMethod*.***POST***, "/alth").permitAll()

Liberar URL de autenticação com a linha de código acima em configure(HttpSecurity http)

5º - private AuthenticationManager authManager

Autenticação programática (manual): atributo adicionado na classe autenticacaoController. Essa classe não vem configurada para injeção de dependência, deve ser feito manualmente na classe SecurityConfiguration.

Nesta classe, devemos sobrescrever o método authenticationManager(), colocando @Bean no método sobrescrito.

6º - Adicionando os dados para autenticação

Na classe controller colcoamos:

UsernamePasswordAuthenticationToken dadosLogin = form.converter() 🡪criar o método converter, que devolve esse objeto, passando email e senha no método.

+ Authentication authentication = authManager.authenticate(dadosLogin)

+ return ResponseEntity.*ok*().build();

7º - authenticate(dadosLogin) pode lançar erro, logo, tratar com try/catch, usando AuthenticationException [Spring Security] para retornar Status 400 (bad request)em caso de erro.

7.2 – Caso OK// Gerar token para devolver no return de status ok

String token = tokenService.gerarToken(authentication): criar classe tokenService com o método. Usa: @Service

7.2.2 – método gerarToken(Authentication authentication)

Usuario logado = (Usuario)authentication.getPrincipal();

Date hoje = new Date();

Date dataExpiracao = new Date(hoje.getTime() + Long.*parseLong*(expiration));

return Jwts.*builder*()

.setIssuer("API Forum ALURA") //remetente do token

.setSubject(logado.getId().toString()) //a quem pretence o token

.setIssuedAt(hoje) //data de criação

.setExpiration(dataExpiracao) //data de expiração do token

.signWith(*SignatureAlgorithm*.***HS256***, secret) //criptografia do token

.compact(); //finalizar sets

7.2.3 – variável private String expiration, com escopo de método

Notação (acima): *@Value*("${forum.jwt.expiration}")

Pega o valor String no destino informado dentro de application.properties

O mesmo ocorre para private String secret

7.2.3.2 - **application.properties**

# jwt

forum.jwt.secret=[valor muito grande-String-sem\_aspas](senha)

forum.jwt.expiration=86400000

8º - return ResponseEntity.*ok*(new TokenDto(token,"Bearer"))~~.build()~~

Criando Dto do token e passando a String “Bearer”, que é o tipo de autenticação (que manda um token junto). Deve-se mudar o tipo de retorno do método para ResponseEntity<TokenDto>, pois agora estamos mandando junto com a mensagem de OK, um TokenDto. Como passamos um objeto, o “build” sai do fim do método.

### Autenticação Via JWT

1º - Authorization = Bearer [token] // no Postman devemos passer essa informação no cabeçalho de requisição.

2º - public class AutenticacaoViaTokenFilter extends OncePerRequestFilter

Criar classe para validar o token que vem na requisição.

filterChain.doFilter(request, response): deve ser a última linha do método exigido.pela SUPER.

2.2 - recuperarToken(HttpServletRequest request): método dentro do método anterior.

String token = request.getHeader("Authorization")

if(token == null || token.isEmpty() || !token.startsWith("Barear "))

return null

return token.substring(7,token.length())

2.3 – Registrar o filtro para o Spring: não é via notação, mas sim na classe SecurityConfiguration (criada neste projeto). A configuração deve ser feita após a linha .sessionManagement().sessionCreationPolicy(*SessionCreationPolicy*.***STATELESS***), adicionando a seguinte linha:

.and().addFilterBefore

(new AutenticacaoViaTokenFilter(), UsernamePasswordAuthenticationFilter.class)

2.4 – Verificar token da requisição Json:

Class AutenticacaoViaTokenFilter (Filter 🡪 extends OncePerRequestFilter) // doFilterInternal([...]) //

A lógica do JsonWebToken está na classe TokenService (que é um Filter) criada separada. Então, passaremos para lá a responsabilidade por fazer a checagem do token (criar método boleano), e nesta classe (autenticacao) ficará o retorno.

🡺 Classes “Filter” criadas pelo Spring não recebem injeção de dependências, não dá pra usar @Autowired. Neste caso, usa-se passar o TokenService como parâmetro no construtor de AutenticacaoViaTokenFilter. Em SecurityConfiguration podemos fazer injeção dependências (por não ser um Filter) e passar como parâmetro no new AutenticacaoViaTokenFilter(\_). Dessa forma resolvemos o problema.

2.4.2 – isTokenValido(String token): Este método fica em TokenService e irá dizer se o token informado é válido.

try {

Jwts.*parser*().setSigningKey(secret).parseClaimsJws(token)

return true

} catch (Exception e) {

return false

x

## 🡺Anotações no código

|  |  |
| --- | --- |
| **Anotação** | **Função** |
| Método  (@RequestParam class nome)  Podemos fazer com que o atributo no método não seja obrigatório:  *@RequestParam*(required = false) | Avisa ao Spring que o parâmetro é obrigatório e vem no endereço de URL. Lança erro 400 em caso de não informar.  A URL não precisa ser alterada, nada de {id} ou coisa do tipo é necessário. Pois aqui se usa uma lógica de query, com “?” na URL informando os parâmetros, logicamente deve ser possível aplicar filtros de consulta dessa forma (como em umas query, de fato). |
| Metodo(@PageableDefault  (sort[atrb],direction[enum]) Pageable paginacao)  Pode colocar também os demais registros: “page" e “size” | Define como deverá ocorrer a ordenação da página por padrão, quando não dor informado na URL |
| @EnableCaching  MAIN class | Habilitar uso de memória caching na aplicação. |
| [import do Spring]  @Cacheable(value)  Método() | O Spring irá guardar o retorno do método em cache. “value” é uma String determinada na hora, serve como um ID para esse dado guardado em memória. |
| @CacheEvict  (value = "idInformado",  allEntries = true)  Método() | Faz com que um método exclua o retorno salvo de um outro método, passando em “value” o ID (String) informado para armazenar o retorno em cachê do outro método. “allEntries” informa se é para excluir todos os registros. |
| @EnableWebSecurity  @Configuration  Class{} | A classe é usada para configurar a segurança co Spring Security |