Sumário

[Introdução 3](#_Toc122551884)

[Objetivos 3](#_Toc122551885)

[Pré requisitos 3](#_Toc122551886)

[Projetos e Softwares 3](#_Toc122551887)

[1. Boas práticas na API 4](#_Toc122551888)

[Utilizando o ResponseEntity do Spring 4](#_Toc122551889)

[Testando retorno do delete 5](#_Toc122551890)

[Corrigindo os outros métodos 5](#_Toc122551891)

[Listar 5](#_Toc122551892)

[Atualizar 6](#_Toc122551893)

[Cadastrar 6](#_Toc122551894)

[Criando o método de detalhar medico 8](#_Toc122551895)

[2. Lidando com erros na API 9](#_Toc122551896)

[Application.propperties 9](#_Toc122551897)

[Retirando o Stacktrace da resposta da requisição 9](#_Toc122551898)

[Organizando pacotes 10](#_Toc122551899)

[Criando classe de tratamento de erros 10](#_Toc122551900)

[404 – Not Found 10](#_Toc122551901)

[400 – Bad Request 11](#_Toc122551902)

[Personalizar mensagens de erro do Bean Validation 12](#_Toc122551903)

[3. Spring Security 13](#_Toc122551904)

[Autenticação e autorização 13](#_Toc122551905)

[Adicionando o Spring Security 15](#_Toc122551906)

[Entidade usuário e migration 17](#_Toc122551907)

[hashing de senha 18](#_Toc122551908)

[Repository e Service 19](#_Toc122551909)

[Montando consultas dinâmicas com Spring Data 20](#_Toc122551910)

[Configurações de segurança 20](#_Toc122551911)

[Reorganizando os pacotes e criando classe de configurações de segurança: 21](#_Toc122551912)

[Controller de autenticação 23](#_Toc122551913)

[Criando senha criptografada: 25](#_Toc122551914)

[Configurando o tipo de criptografia 26](#_Toc122551915)

[Implementando a interface UserDetails em nossa entidade de Login 26](#_Toc122551916)

[Testando autenticação: 28](#_Toc122551917)

[4. Json web Token 29](#_Toc122551918)

[Buscando a dependência 29](#_Toc122551919)

[Criando o serviço de geração de token 29](#_Toc122551920)

[Ajustes na geração do token 32](#_Toc122551921)

[5. Controle de acessos 33](#_Toc122551922)

[Interceptando requisições 33](#_Toc122551923)

[Criando o filter de segurança 34](#_Toc122551924)

[Recuperando o token 35](#_Toc122551925)

[Validando o token recebido 37](#_Toc122551926)

[Autenticando o usuário 38](#_Toc122551927)

[Última configuração: 40](#_Toc122551928)

# Introdução

## Objetivos

* Boas práticas na API
  + Protocolo HTTP
  + Retornos corretos
* Tratamento de erros
  + O que fazer quando a aplicação levanta uma exceção
* Autenticação/Autorização
  + Segurança
* Tokens JWT

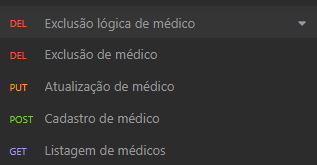
# Pré requisitos

## Projetos e Softwares

Como esse projeto é uma continuação direta do curso *Spring Boot 3 - desenvolva uma API Rest em Java*, os requisitos se mantém, além de ter o projeto do curso anterior.

# 1. Boas práticas na API

No curso anterior montamos uma API de que cadastra, lê, atualiza e apaga dados, e nela, anotamos corretamente os verbos HTTP:

* Remover: delete
* Cadastrar: post
* Listar: get
* Atualizar: put

Porém, não são somente os verbos do protocolo HTTP e a URL que consideramos importantes, precisamos ter um tratamento do retorno da API nas requisições.

O protocolo HTTP possui diversos códigos para vários cenários e não estamos usando esse recurso de forma adequada. Estamos devolvendo 200 quando dá certo ("OK"), ou 500 que o Spring devolve de forma automática, ou erro 400 caso ocorra um erro de validação do bean validation.

## Utilizando o ResponseEntity do Spring

ResponseEntity é uma classe que nos fornece vários tipos de retornos e nos permite personalizar os mesmo.

Por não colocar um retorno específico estamos deixando o Spring tomar conta dos retornos, devolvendo 200, 400 ou 500.

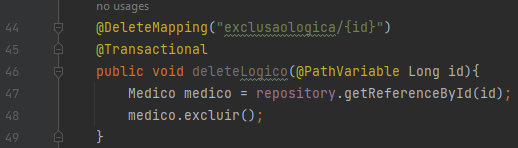


Figura 1 Metodo delete do tipo void

Por exemplo, no método excluir, o mais adequado seria devolver o código 204, que se refere à requisição processada e sem conteúdo.

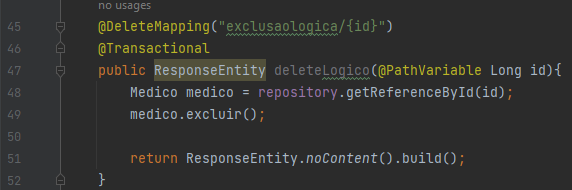
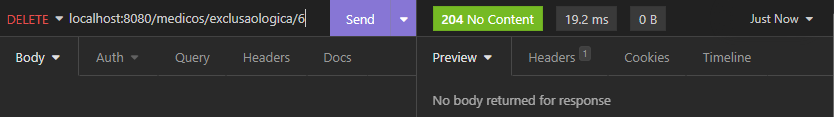


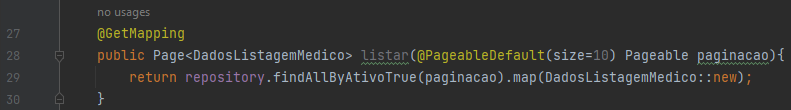
Figura 2 Metodo delete com o retorno correto seguindo o protocolo HTTP

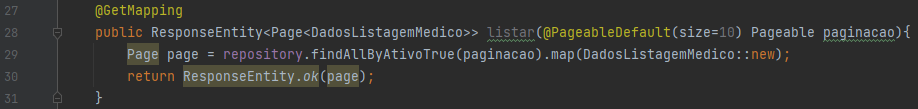
## Testando retorno do delete

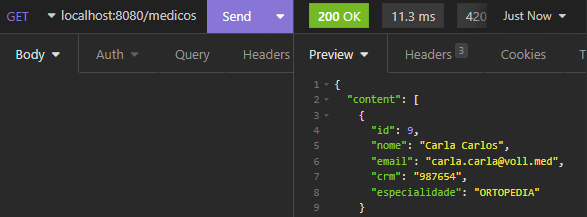


## Corrigindo os outros métodos

## Listar



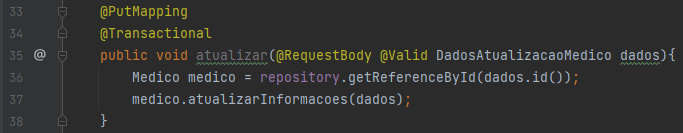




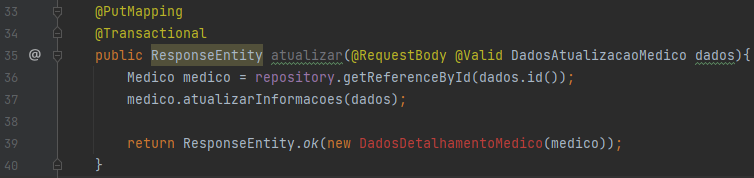
Nesse caso “não” mudou nada, já que no protocolo HTTP diz que para buscar dados temos que mandar o código 200 e o objeto. Mas assim utilizamos as boas práticas utilizando o ResponseEntity.

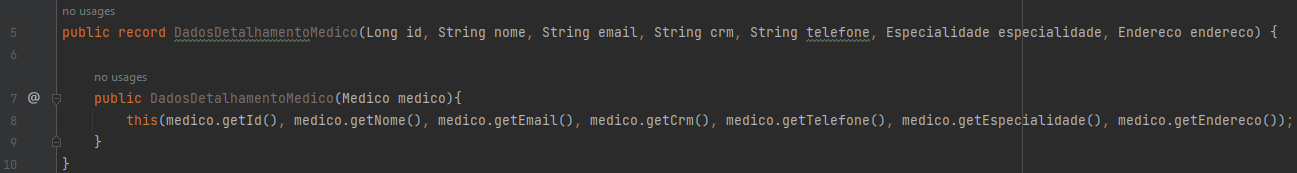
## Atualizar

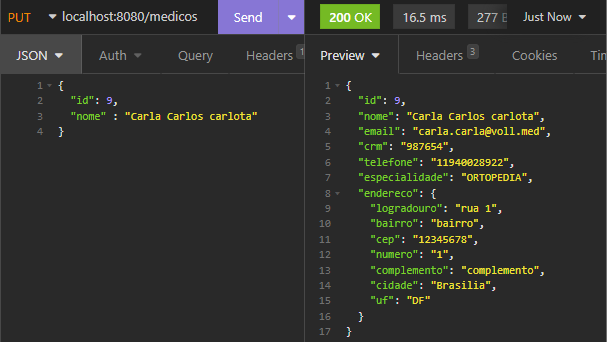
Bem semelhante ao excluir, vamos atualizar os dados, devolver 200 mas devolver o objeto recém atualizado.



Não é recomendado passar a entidade JPA no response, então vamos criar um DTO DadosDetalhamentoMedico para fazer essa intermediação.



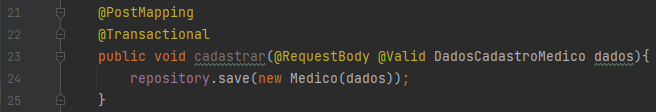




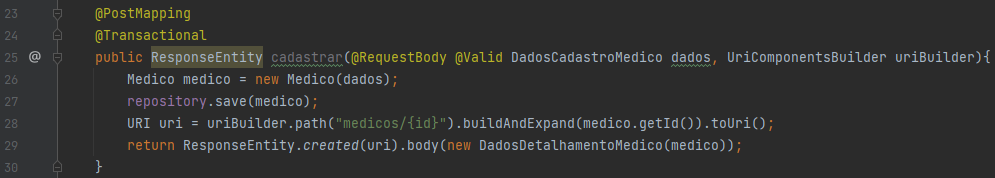
## Cadastrar

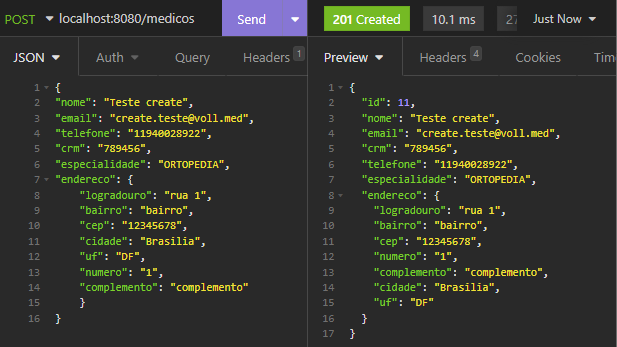
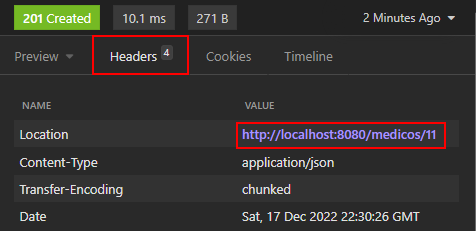
Conforme o protocolo HTTP, ao cadastrarmos uma informação ou recurso em uma API, o código HTTP que deve ser devolvido, neste cenário, é o código 201 chamado created. Esse código significa que um registro foi criado na API.

Código 201: devolve no corpo da resposta os dados do novo recurso/registro criado e um cabeçalho do protocolo HTTP (Location). Esse cabeçalho mostra o endereço para que o front-end, ou aplicativo mobile consiga acessar o recurso cadastrado. Logo, no cadastro não devolvemos apenas o código 200 OK e nem apenas o 201.



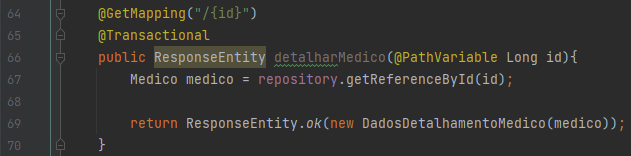
O UriComponentBuilder será responsável por pegar o endereço de nossa aplicação (exemplo: <https://localhost:8080/>), assim não importando onde implantarmos garantiremos que pegará a URI correta.

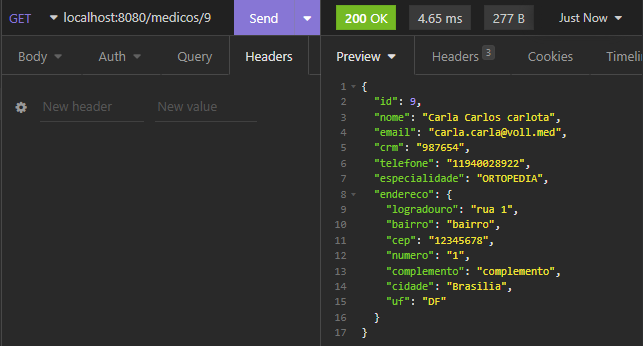


Dessa forma, temos o cabeçalho location com o endereço http://localhost:8080/medicos/11, sendo o número 11 o ID do médico que acabamos de cadastrar no banco de dados.

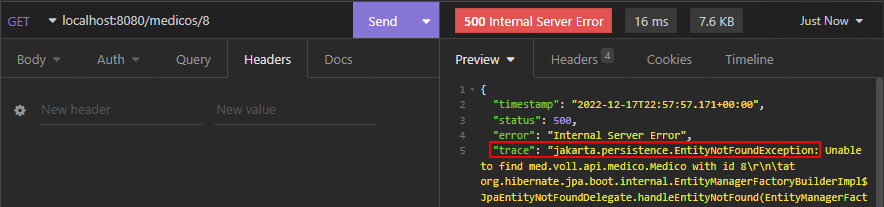
## Criando o método de detalhar medico





# 2. Lidando com erros na API

Quando fazemos uma requisição de um médico que não existe em nossa base temos como resposta o código 500, erro no servidor, com toda a stacktrace, e isso, pode expor brechas de segurança.

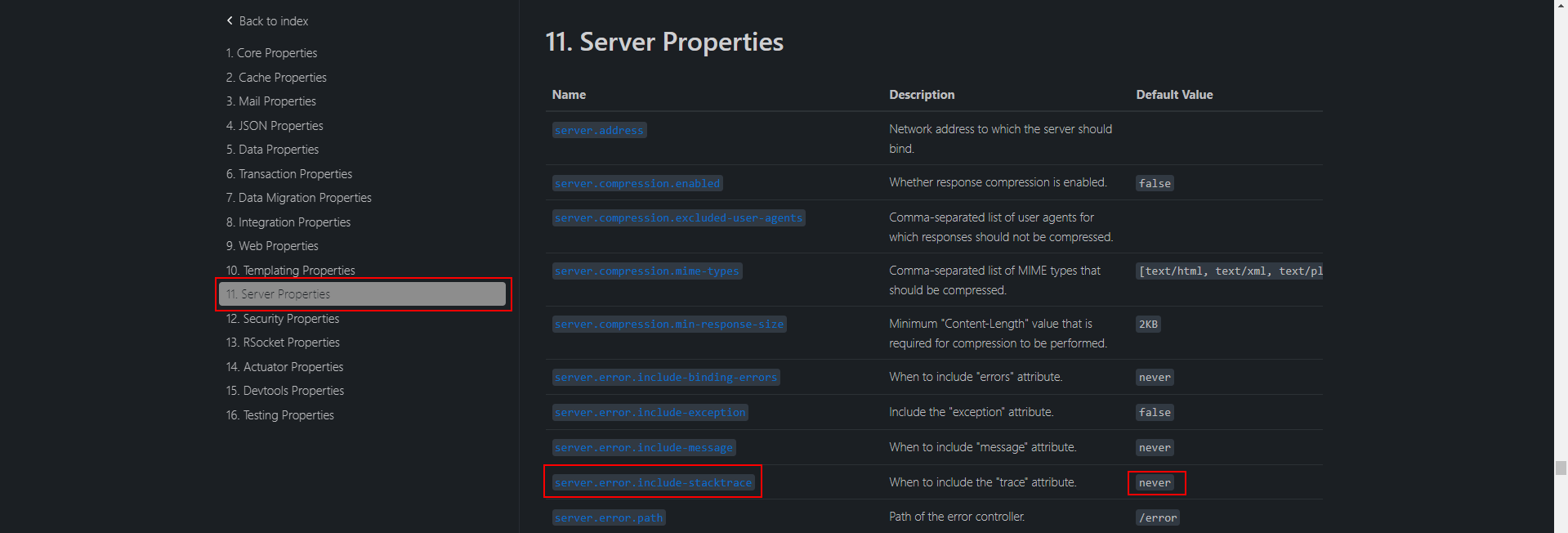


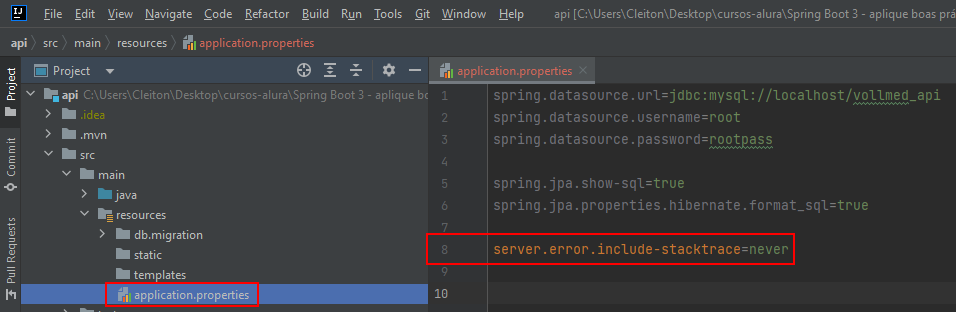
## Application.propperties

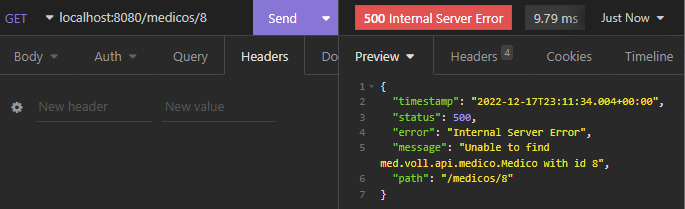
Nele é onde definimos configurações do Spring em nosso projeto. Para buscar alguma configuração, basta pesquisar por Spring boot properties.

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html>

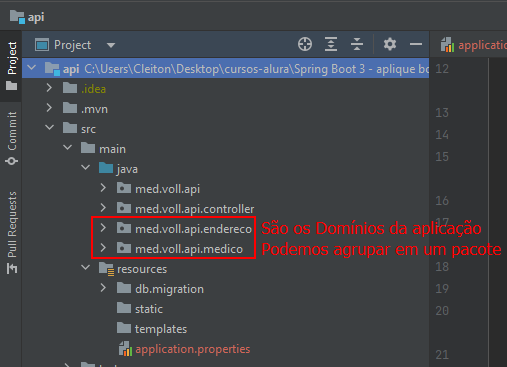
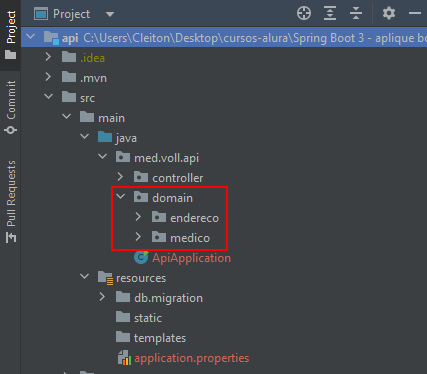
## Retirando o Stacktrace da resposta da requisição





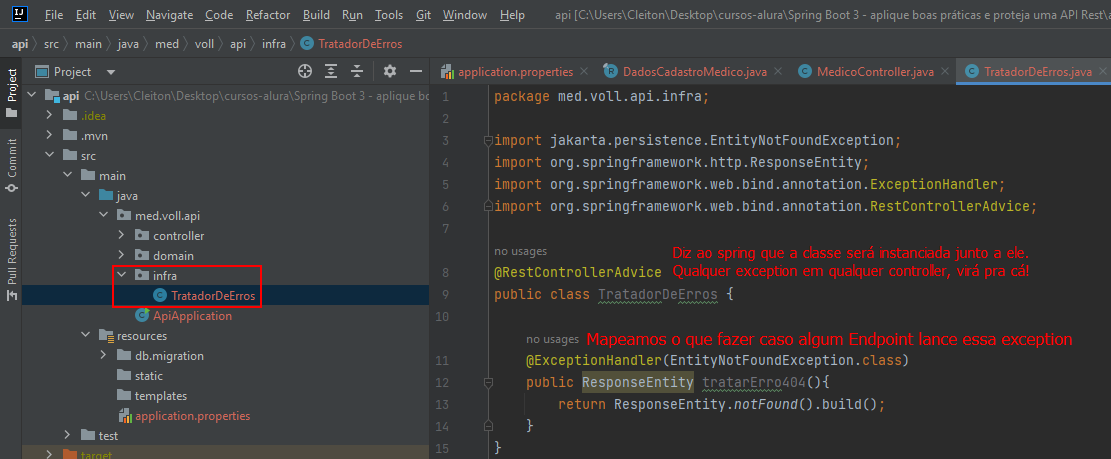


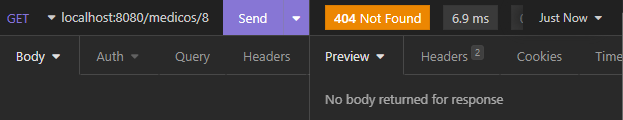
## Organizando pacotes

## Criando classe de tratamento de erros

## 404 – Not Found

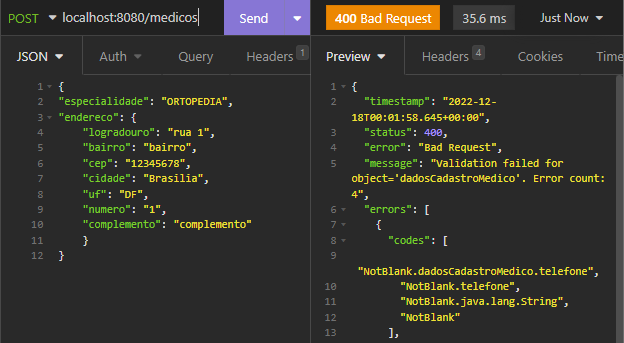




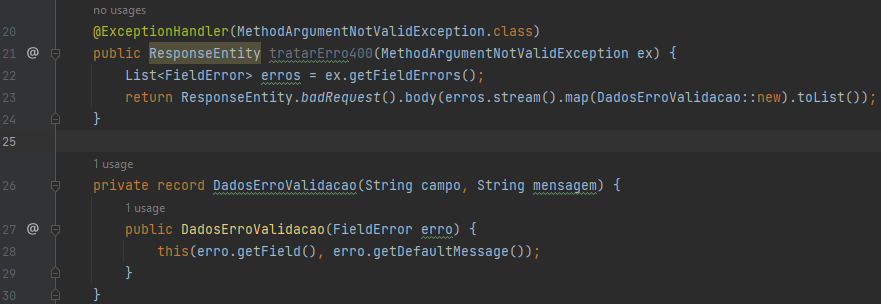
## 400 – Bad Request

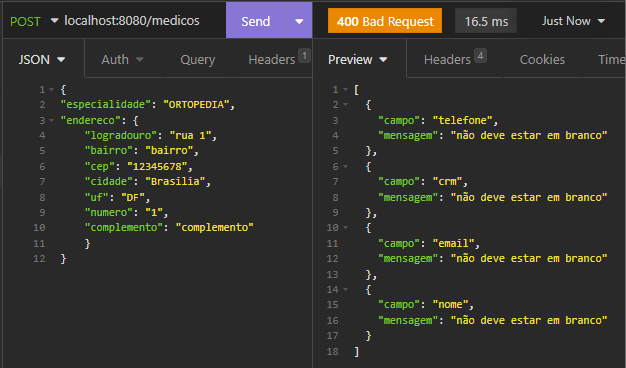
Indica que o servidor não conseguiu processar uma requisição por erro de validação nos dados enviados pelo cliente.

O Spring, por padrão, devolve uma lista enorme com muitas informações desnecessárias.



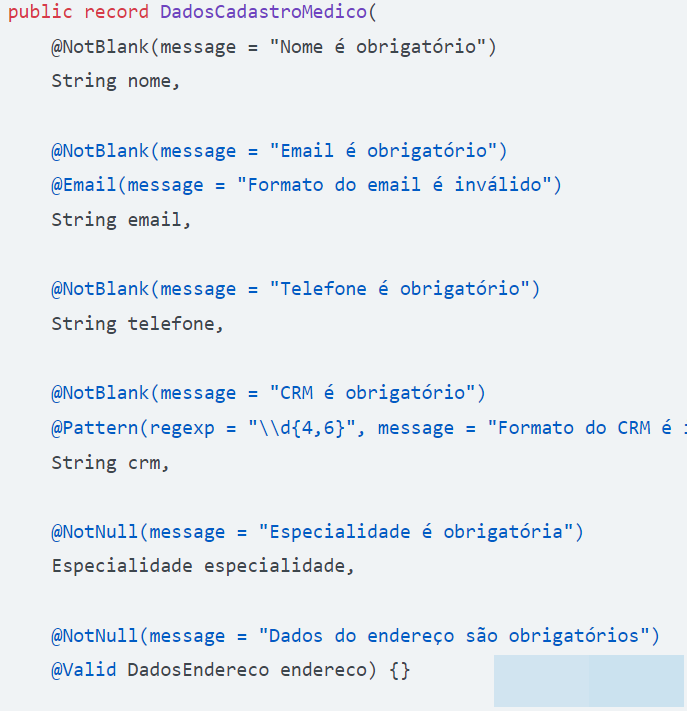
Quando fazemos uma requisição faltando algum campo a exceção que é lançada é MethodArgumentNotValidException, a mesma que vem com todas aquelas informações. Vamos criar um DTO provado, já que vamos utilizar apenas aqui, e fazer a conversão para devolver



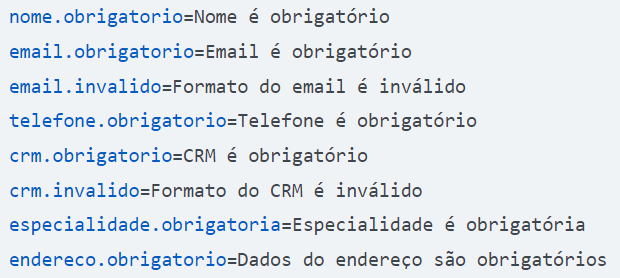


## Personalizar mensagens de erro do Bean Validation

Uma das maneiras de personalizar as mensagens de erro é adicionar o atributo message nas próprias anotações de validação:



Outra maneira é isolar as mensagens em um arquivo de propriedades, que deve possuir o nome ValidationMessages.properties e ser criado no diretório src/main/resources



E, nas anotações, indicar a chave das propriedades pelo próprio atributo message, delimitando com os caracteres { e }:



# 3. Spring Security

## Autenticação e autorização

O Spring contém um módulo específico para tratar de segurança, conhecido como Spring Security.

**Objetivos do Spring Security**

* Autenticação
* Autorização (controle de acesso)
* Proteção contra-ataques (CSRF, clickjacking, etc.)

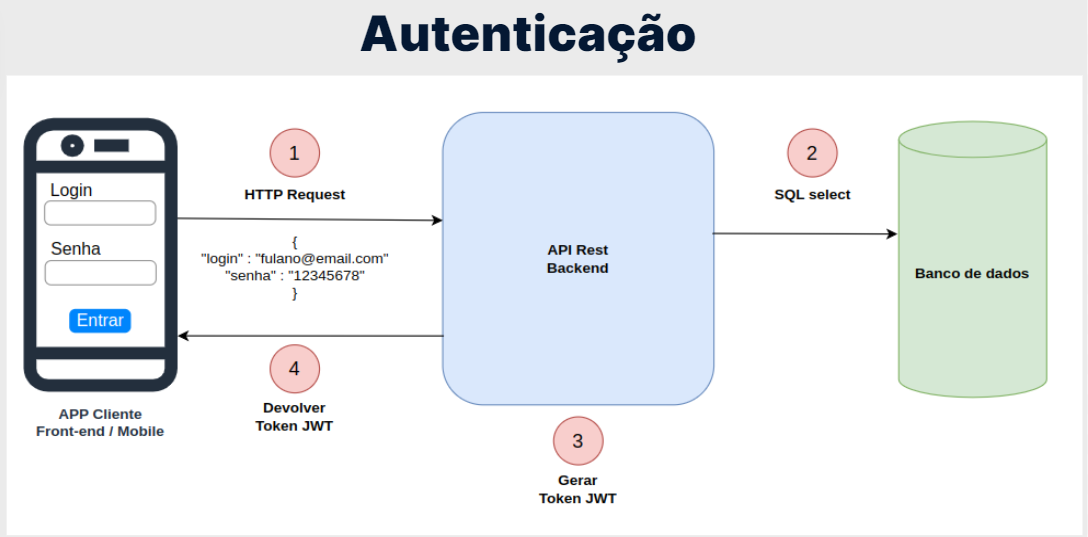
A API back-end não deve ser pública, ou seja, receber requisições sem um controle de acesso. A partir disso, entra o Spring Security para nos auxiliar na proteção dessa API no back-end.

ATENÇÃO: Autenticação em aplicação Web (Stateful) != Autenticação em API Rest (Stateless).

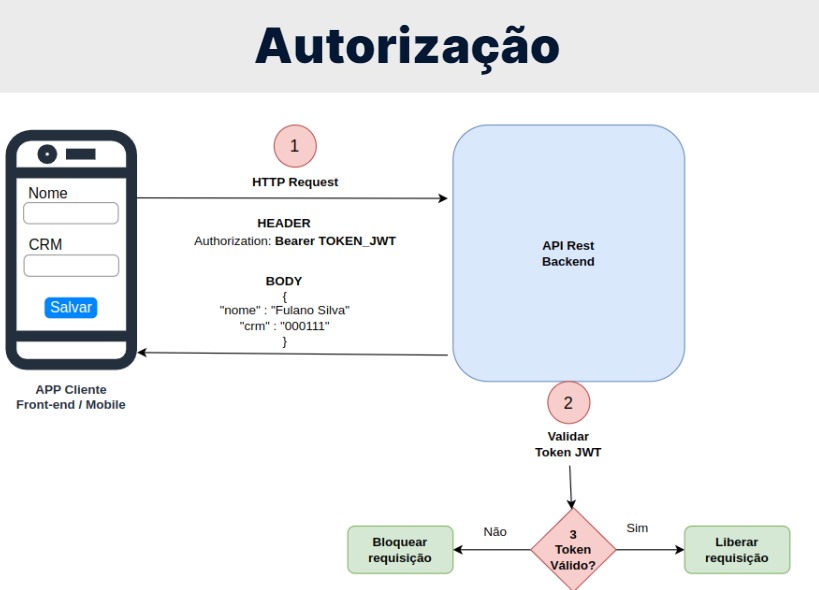
O processo de autenticação em uma aplicação Web tradicional é diferente do processo de autenticação em uma API Rest. Em uma aplicação Web, temos um conceito chamado de stateful.

Toda vez que um usuário efetua o login em uma aplicação Web, o servidor armazena o estado. Isto é, cria as sessões e, com isso, consegue identificar cada usuário nas próximas requisições.

Como será o processo de autenticação em uma API? Temos diversas estratégias para lidarmos com a autenticação. Uma delas é usando Tokens, e usaremos o JWT - JSON Web Tokens como protocolo padrão para lidar com o gerenciamento desses tokens - geração e armazenamento de informações nos tokens.



Esse token deve ser armazenado pelo aplicativo mobile/front-end. Há técnicas para guardar isso de forma segura, porque esse token que identifica se o usuário está logado.



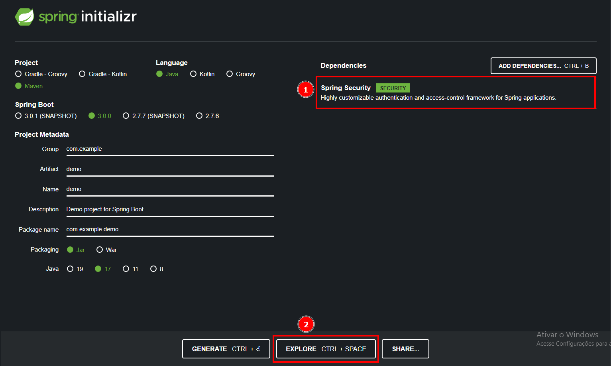
Será disparada uma requisição para a nossa API - da mesma forma que fizemos no Insomnia. No entanto, além de enviar o JSON com os dados do médico no corpo da resposta, a requisição deve incluir um cabeçalho chamado authorization. Neste cabeçalho, levamos o token obtido no processo anterior, de login.

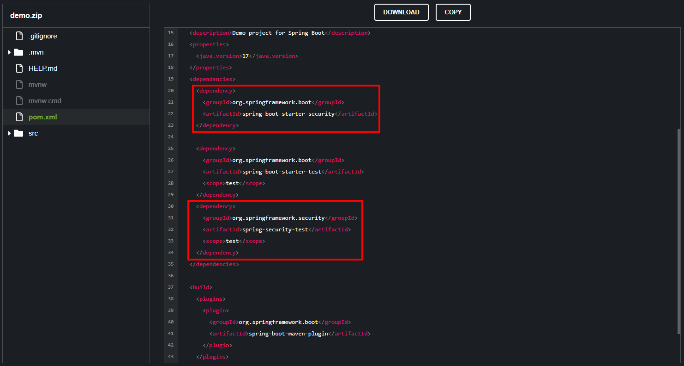
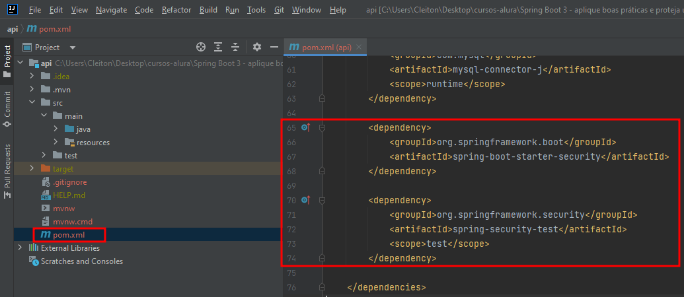
Você pode conferir as principais formas de autenticação lendo este artigo: <https://www.alura.com.br/artigos/tipos-de-autenticacao>

## Adicionando o Spring Security

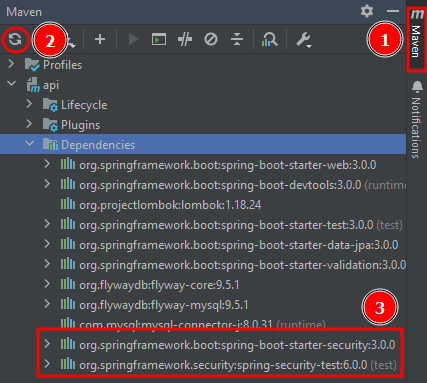
Adicionando as dependências do Spring Security: <https://start.spring.io/>

Acessando o Spring initializr e pegado a dependência do Spring Security

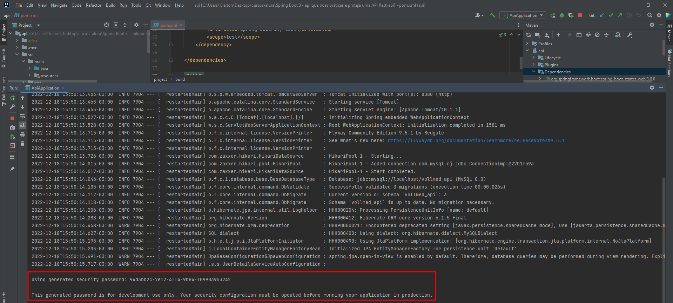


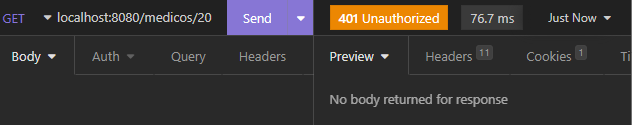
 

Recarregando o projeto para baixar as dependências:



Executando o projeto e validando o Spring Security:





Analisando os logs, perceba que no final foi gerado uma linha que antes não era exibida no retorno.

Using generated security password: 94d3bb2c-7e12-4ff4-9be4-f09903953242

Voltaremos ao Insomnia para disparar a requisição de listagem de médicos. No endereço, temos:

401 Unauthorized, e em "Preview", temos a mensagem: "No body returned for response" ("Nenhum corpo retornou para resposta").

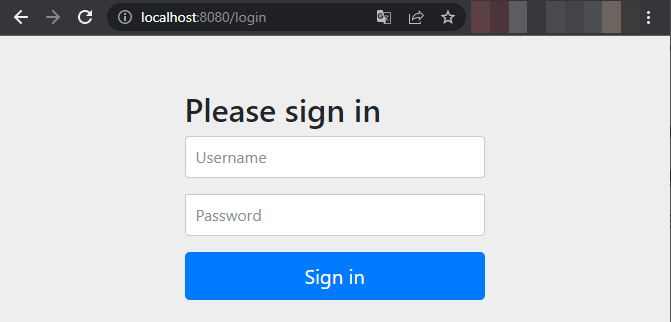
Todas as requisições estão sendo bloqueadas pelo Spring Security. Portanto, esse é o comportamento padrão do Spring Security ao adicioná-lo como dependência no projeto, ele bloqueia tudo e gera uma senha aleatória.

Através do navegador, acessando o Get de listagem de médicos, somos redirecionados para um Formulário de Login gerado pelo próprio Spring.

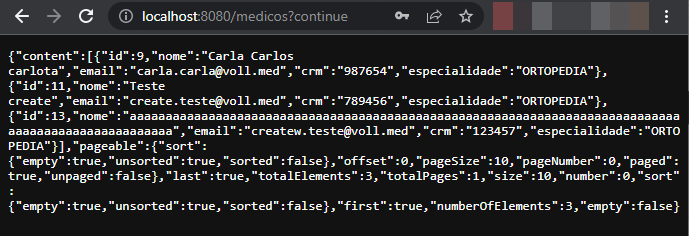
Username: user

Password: 94d3bb2c-7e12-4ff4-9be4-f09903953242

Obs.: Lembrando que cada vez que reiniciarmos a aplicação será gerada uma senha diferente.



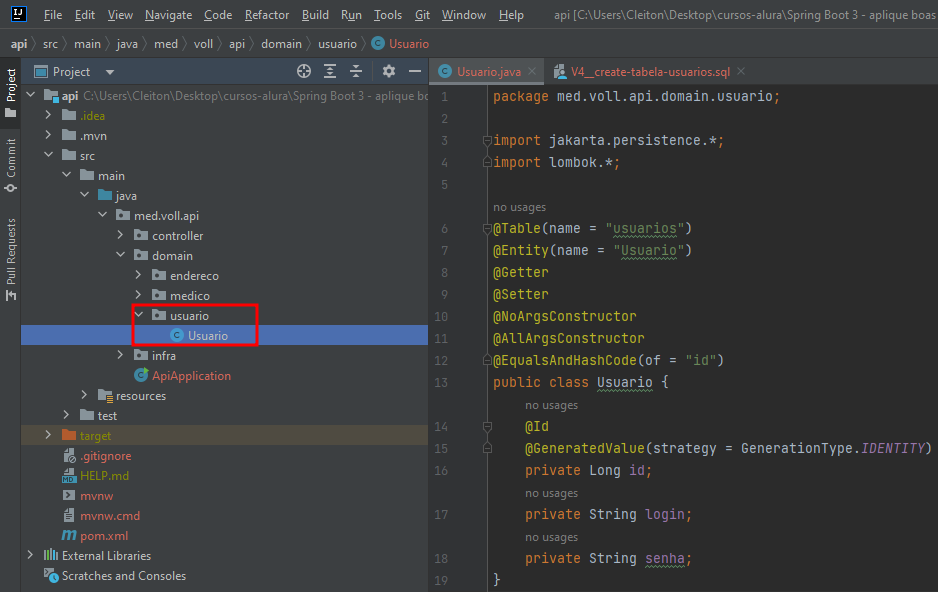
Após o login:



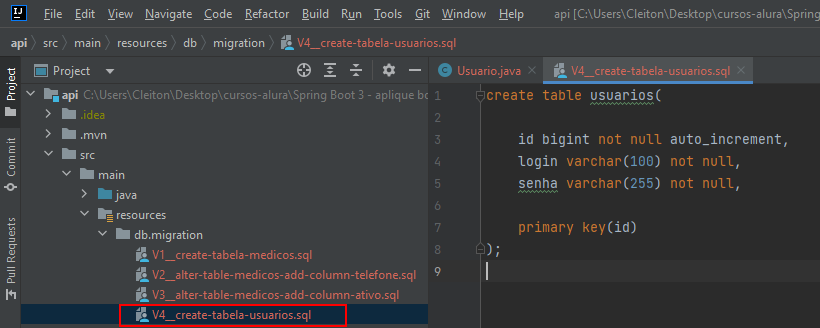
Portanto, esse é o comportamento padrão do Spring Security: ele bloqueia todas as URLs, disponibiliza um formulário de login - que e possui um usuário padrão chamado user e a senha devemos copiar do console ao inicializar o projeto.

## Entidade usuário e migration

Criando entidade de login:

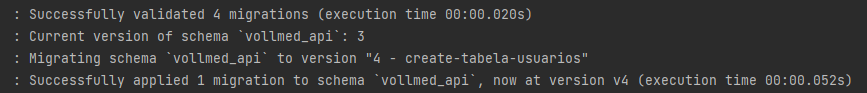


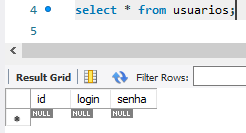
Migration:



Ao inserir um usuário, no campo "login" digitamos o e-mail do usuário (informação única) e em "senha", não vamos armazenar na tabela "123456", isto é, em texto explícito. Usaremos um algoritmo de hash.

Executando o projeto...





## hashing de senha

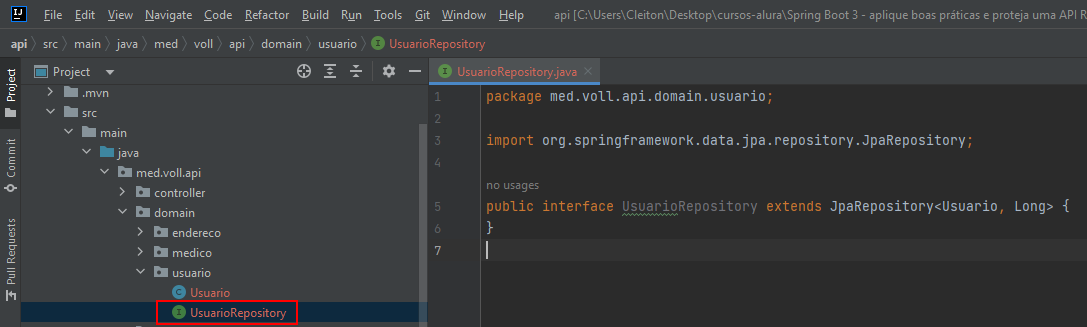
Senhas são informações sensíveis e não devem ser armazenadas em texto aberto, pois se uma pessoa mal intencionada conseguir obter acesso ao banco de dados, ela conseguirá ter acesso às senhas de todos os usuários. Para evitar esse problema, você deve sempre utilizar algum algoritmo de hashing nas senhas antes de armazená-las no banco de dados.

Existem diversos algoritmos de hashing que podem ser utilizados para fazer essa transformação nas senhas dos usuários, sendo que alguns são mais antigos e não mais considerados seguros hoje em dia, como o MD5 e o SHA1. Os principais algoritmos recomendados atualmente são:

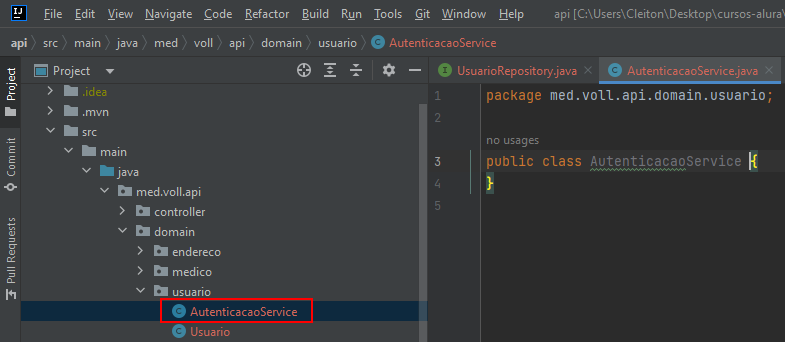
* Bcrypt
* Scrypt
* Argon2
* PBKDF2

## Repository e Service

UsuarioRepository para persistência no banco de dados

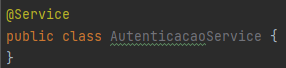


**Vamos criar algumas classes de configurações para o Spring Security**

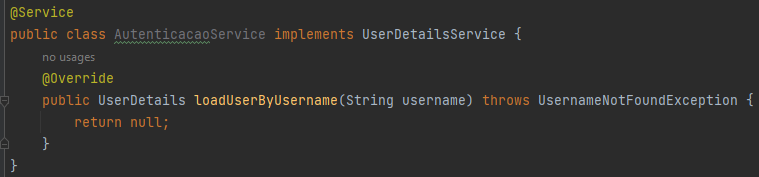


O Spring, quando iniciado, procura classes com alguma implementação automaticamente, mas para ele identificar usamos algumas anotações.

Para o Spring identificar a nossa classe de autenticação precisamos anotar com @Service. A anotação @Service serve para o Spring identificar essa classe como um componente do tipo serviço, no caso, será o de autenticação.

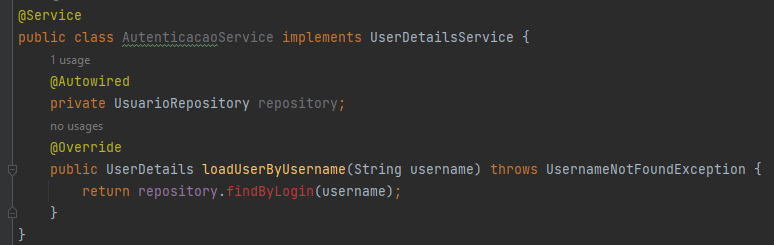


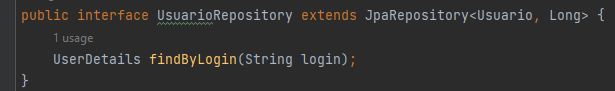
Para informar ao Spring Security que essa é uma classe de serviço de autenticação precisaremos implementar uma interface, a UserDetailsService, que é uma interface do Spring Security.



Com isso, quando o usuário efetuar o login, o Spring busca pela classe AutenticacaoService - por ser a responsável por implementar a UserDetailsService - e chama o método loadUserByUsername, passando o username digitado no formulário de login.

Como essa implementação faz a autenticação pelo nome de usuário, precisamos buscar no banco com o nosso repository.





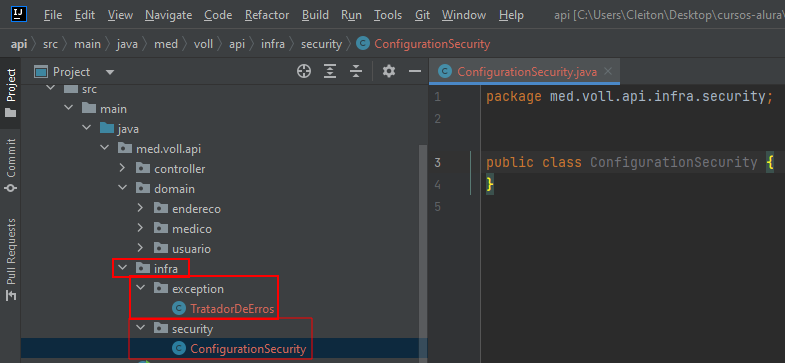
## Montando consultas dinâmicas com Spring Data

Para conhecer mais detalhes e entender melhor como montar consultas dinâmicas com o Spring Data, acesse a sua documentação oficial: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

## Configurações de segurança

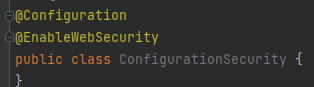
A próxima alteração é configurar o Spring Security para ele não usar o processo de segurança tradicional, o stateful. Como estamos trabalhando com uma API Rest, o processo de autenticação precisa ser stateless.

## Reorganizando os pacotes e criando classe de configurações de segurança:



Criaremos nossa classe de configurações de segurança, onde vamos personalizar as configurações do Spring Security.

Usaremos as anotações @Configuration para o Spring identifica a classe e a carrega no projeto e a @EnableWebSecurity para informarmos ao Spring que vamos **personalizar** as configurações de segurança.



Dentro da classe, vamos incluir a configuração do processo de autenticação, que precisa ser stateless. Criaremos o método que retorna um SecurityFilterChain, do próprio Spring. Ele é usado para configurar o processo de autenticação e de autorização.

O método receberá como parâmetro um HttpSecurity, também do Spring.

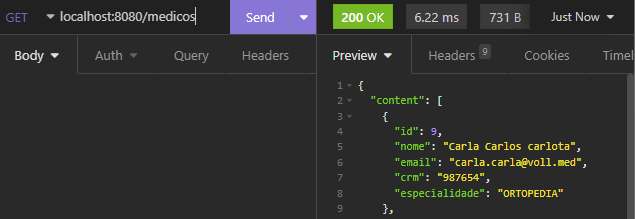
Iremos usar o http para definir as seguintes configurações:

* http.csrf().disable() : Serve para desabilitarmos proteção contra-ataques do tipo CSRF (Cross-Site Request Forgery), que já mitigamos por usar tokens.
* .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS): autenticação para ser stateless, e não usar a tela de login padrão do Spring

Spring não lê o método de forma automática, precisamos incluir a anotação @Bean, que serve exibir o retorno desse método, que estamos devolvendo um objeto SecurityFilterChain.

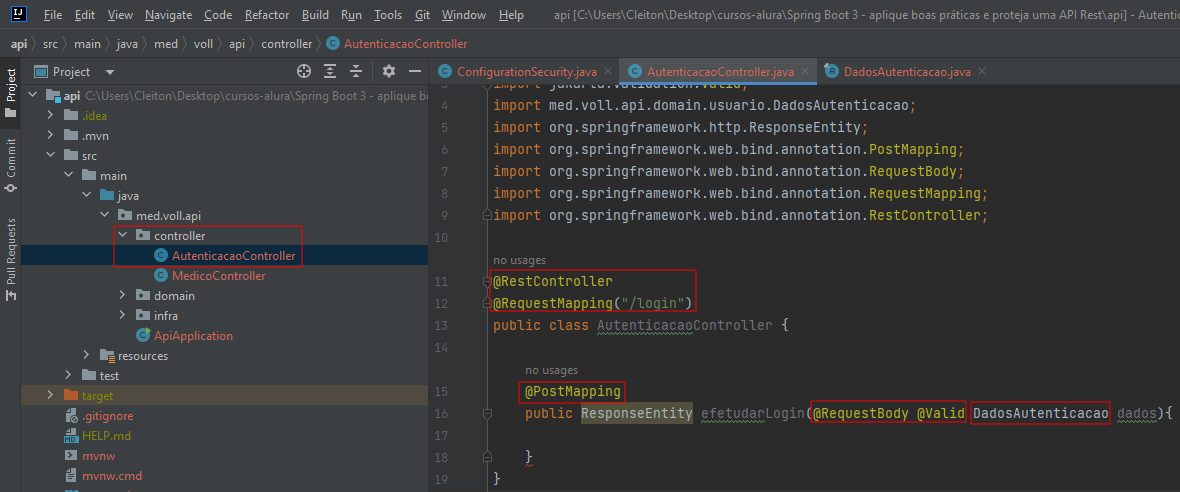


Agora, configuramos para o processo de autenticação ser stateless. Não será mais gerado o formulário de login e senha, quem faz isso é a nossa aplicação mobile, no front-end. E, também, não bloqueará mais a URL.



## Controller de autenticação

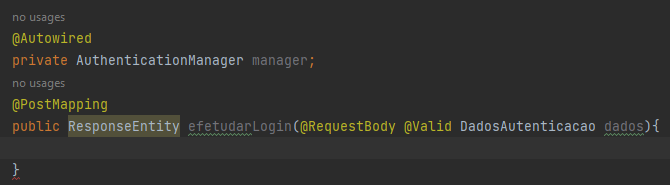
Classe controller:



**DTO de autenticação:**



O processo de autenticação está na classe AutenticacaoService, onde fazemos a consulta do usuário, mas não iremos chama-la no controller diretamente. Para isso vamos utilizar a Classe AuthenticationManager.



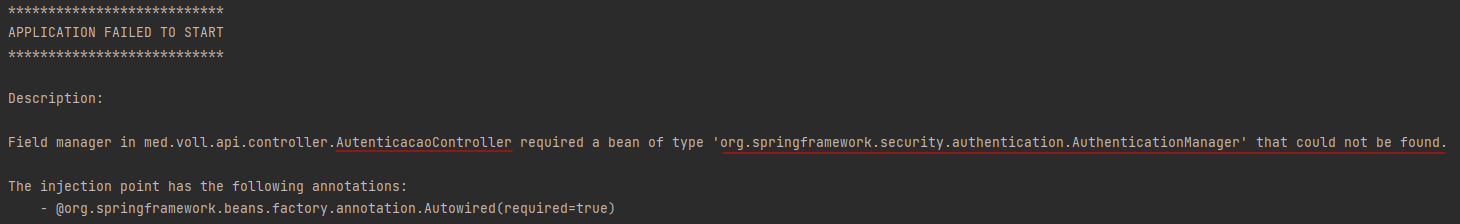
A classe AuthenticationManager tem um método de autenticação que recebe um token, sendo esse token o login e a senha.



Para criar esse token usaremos a classe UsernamePasswordAuthenticationToken passando como parâmetro o DTO, dados.login() e dados.senha().



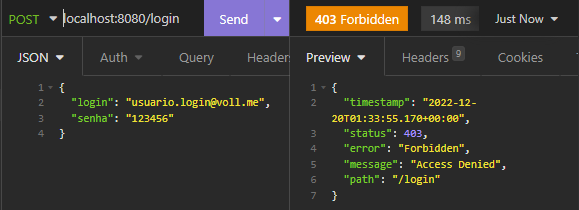
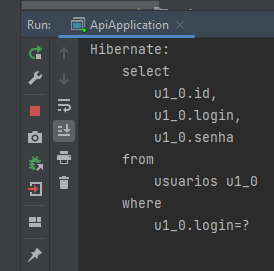
Ao executar o projeto a seguinte mensagem de erro é exibida:



Dizendo que a o campo manager na classe AutenticacaoController requer um bean do tipo AuthenticationManager que não pode ser encontrado. Ou seja, o Spring não sabe instanciar a classe AuthenticationManager, portanto precisamos configurar para ele reconhecer lá na SecurityConfiguration.



Pronto! Bora testar o login.

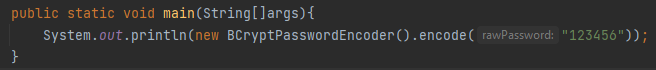
Vemos que a requisição foi feita, voltou acesso negado e nossa aplicação fez a consulta no banco. Isso aconteceu pois não temos nenhum usuário cadastrado.

Vamos cadastrar um usuário, mas NÃO como mostado abaixo, pois não podemos guardar a senha sem criptografia em nosso banco de dados.

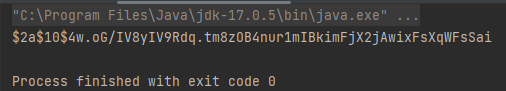


## Criando senha criptografada:

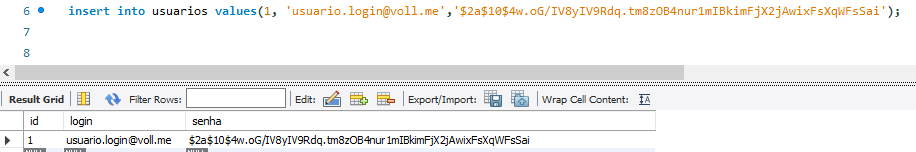
Em qualquer classe do seu projeto crie um método main, APAGUE DEPOIS, e gere a senha usando a classe: BCryptPasswordEncoder



Resultado:



Agora temos nossa senha 123456 criptografada.



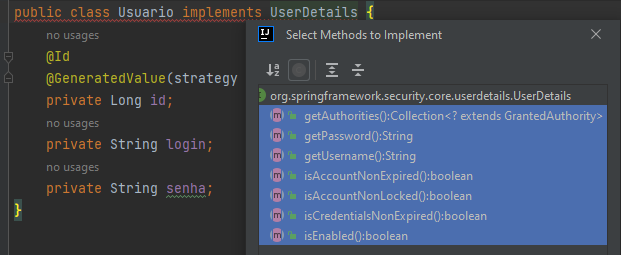
## Configurando o tipo de criptografia

Em nossa classe de configuração SecurityConfiguration, vamos incluir um novo método que diz ao Spring qual é o tipo de criptografia utilizado. Esse método precisa devolver o PasswordEncoder do próprio Spring Security.

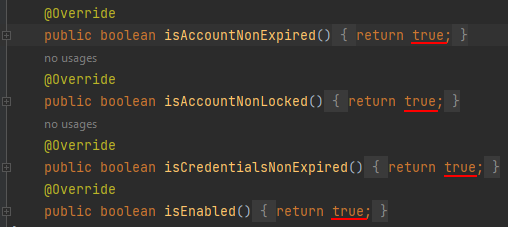


## Implementando a interface UserDetails em nossa entidade de Login

Para o Spring entender quais campos na entidade de Login é a senha e login, precisamos implementar a interface UserDetails.

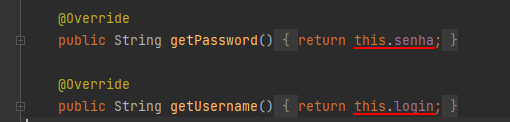


Primeiro vamos trocar todos os métodos que devolvem boolean de false para true.

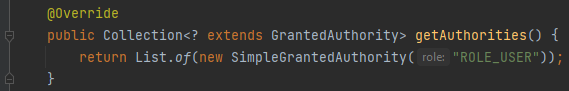


Aqui que controlamos se a conta do usuário tiver uma data de expiração, se pode ser bloqueado, credenciais bloqueadas. Não usaremos, mas caso for implementar, é só criar os atributos na entidade e banco de dados e devolver para os respectivos métodos.

Agora dizemos quais campos é referente ao login e senha:

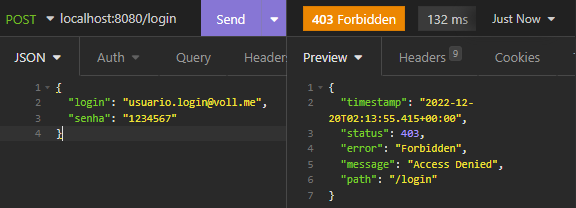


Por fim, o controle de perfis (adm, moderador, usuário comum...), que não utilizaremos, mas o Spring pede que devolvemos algo:

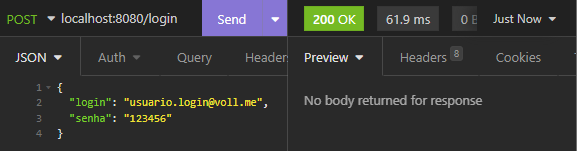


## Testando autenticação:

Senha errada:



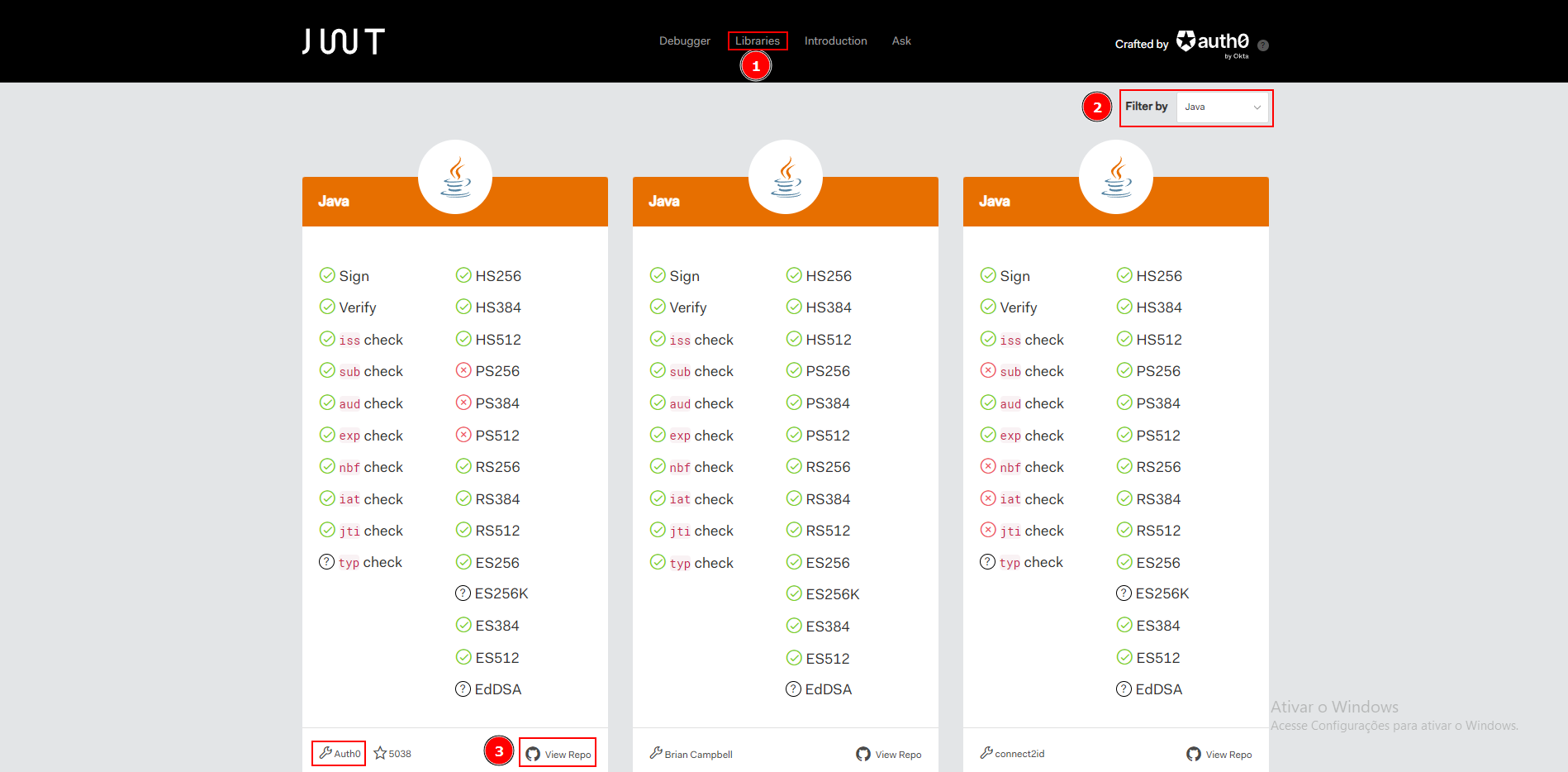
Senha correta:

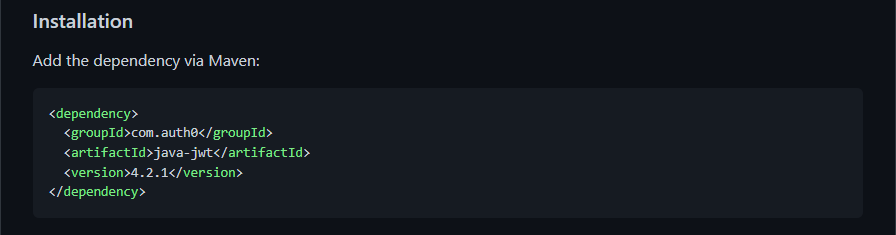


# 4. Json web Token

## Buscando a dependência

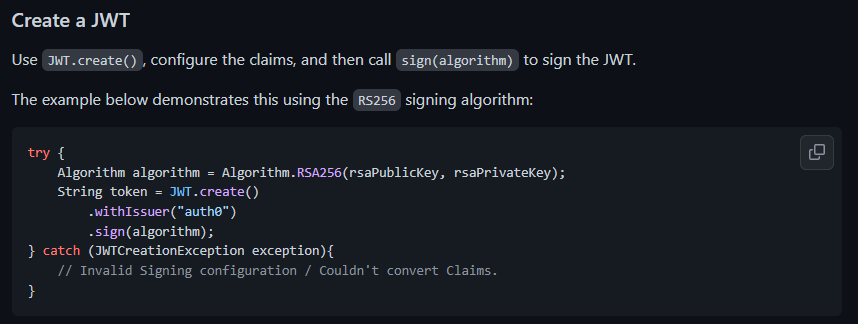
Para pegarmos a biblioteca, acessaremos o site https://jwt.io/. Clicaremos na segunda opção do menu superior do site, "Libraries". Lá, encontraremos uma lista com várias bibliotecas que geram tokens no padrão JWT.



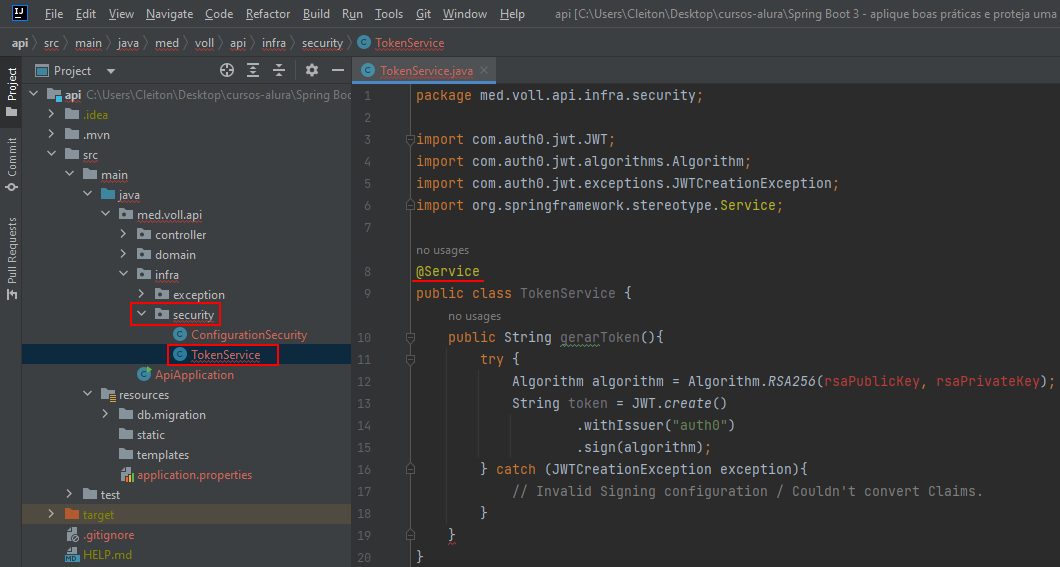


## Criando o serviço de geração de token

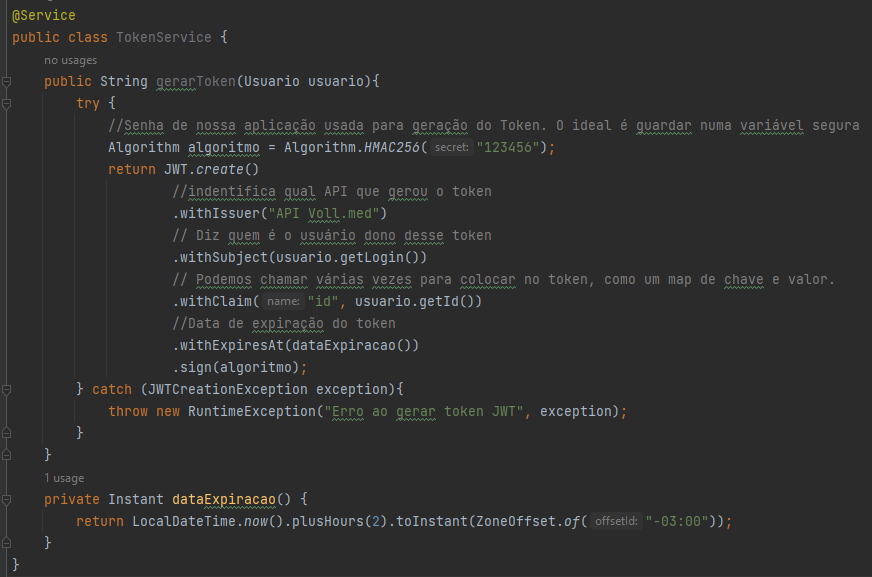
Na documentação no Git do JWT tem um exemplo de como gerar o token



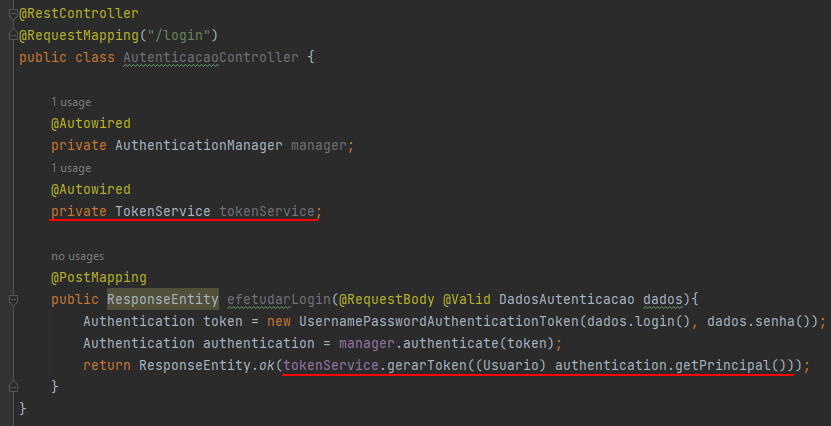
Vamos criar uma classe chamada TokenService e implementar o código de geração do JWT.

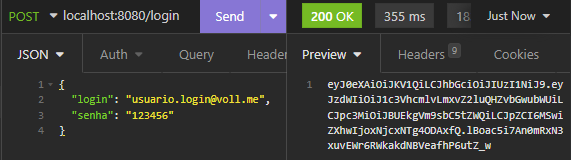


Colocando as configurações

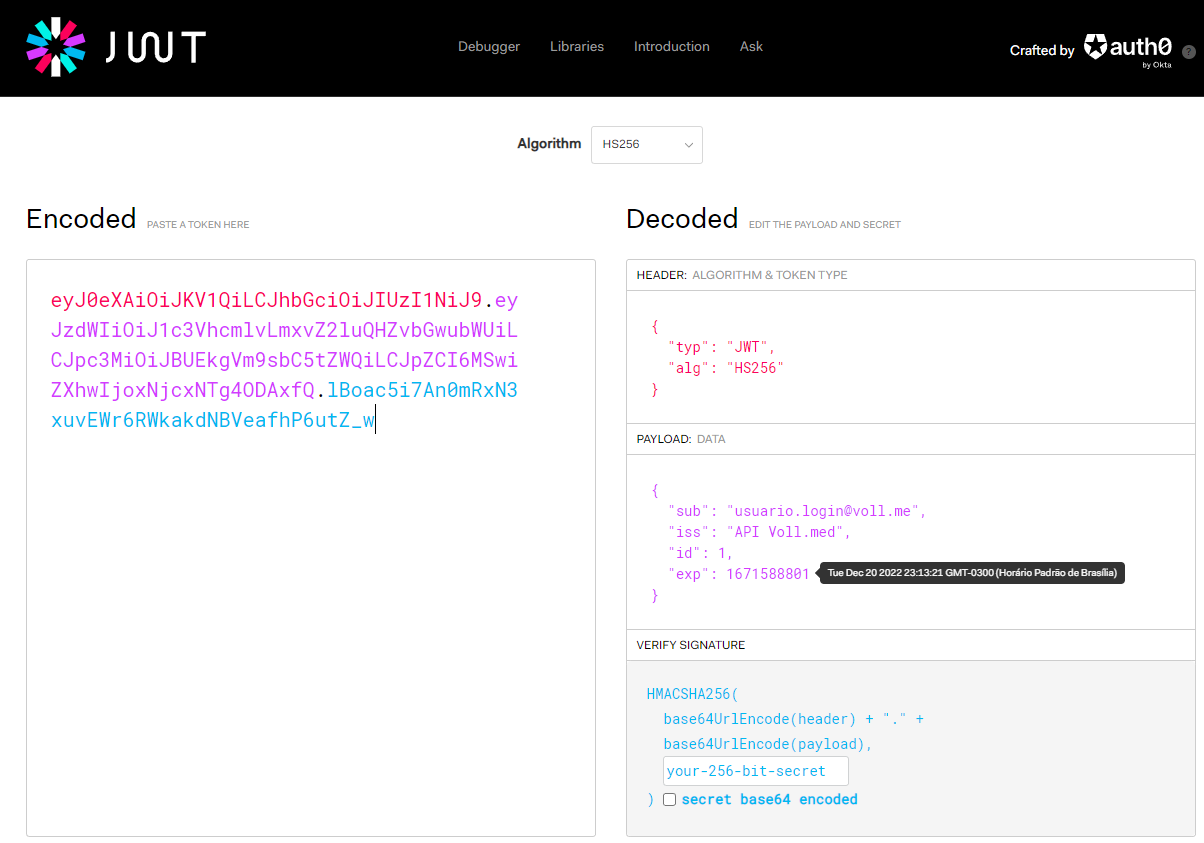


Adicionando geração do token no Controller de login:





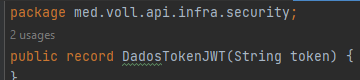
Analisando Token no JWT.io



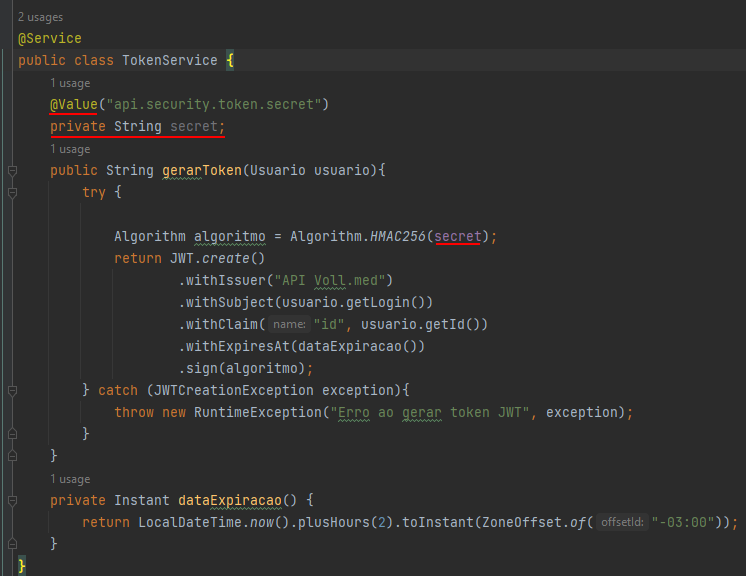
## Ajustes na geração do token

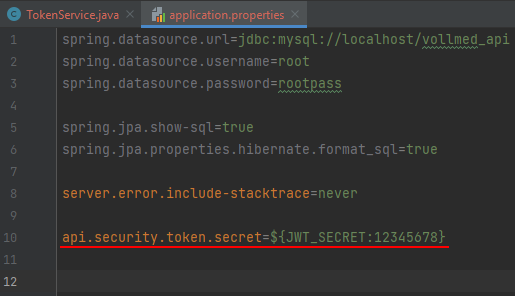
Criando DTO de token:





Colocando secret da geração do token em variáveis de ambiente:





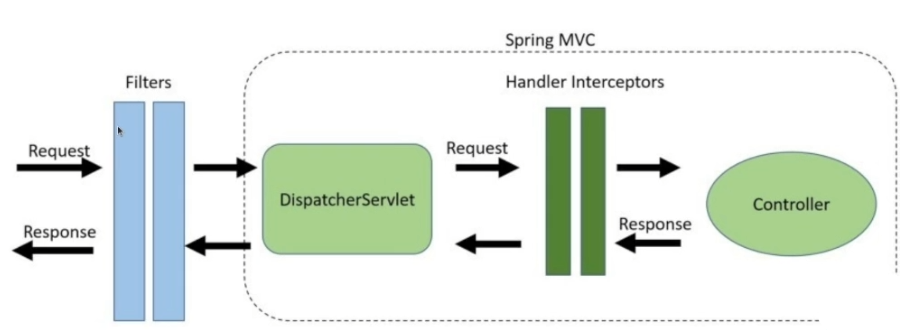
# 5. Controle de acessos

Antes de fazer qualquer requisição precisamos validar antes se o token que está sendo passado é um token válido, mas encontramos alguns problemas:

* Onde vamos validar o token?
* Colocar um If em cada método mapeado em nossa API?

Por isso iremos utilizar a interceptação, que irá pegar TODAS as requisições antes mesmo de chamar qualquer um de nossos métodos.

## Interceptando requisições



O Spring tem uma classe chamada DispatcherSevlet, responsável por receber todas as requisições do projeto. Ela descobre qual controller será preciso chamar em cada requisição.

Depois que a requisição passa pelo DispatcherSevlet, os Handler Interceptors são executados. Com ele, identificamos o controller a ser chamado e outras informações relacionadas ao Spring.

Já os filters aparecem antes mesmo da execução do Spring, onde decidimos se a requisição será interrompida ou se chamaremos, ainda, outro filter.

Portanto, precisaremos criar um filter ou um interceptor no nosso projeto, para que o código, com a validação do token, sejam colocado dentro deles. Ele terá, então, o papel de ser executado como o "interceptador" da requisição.

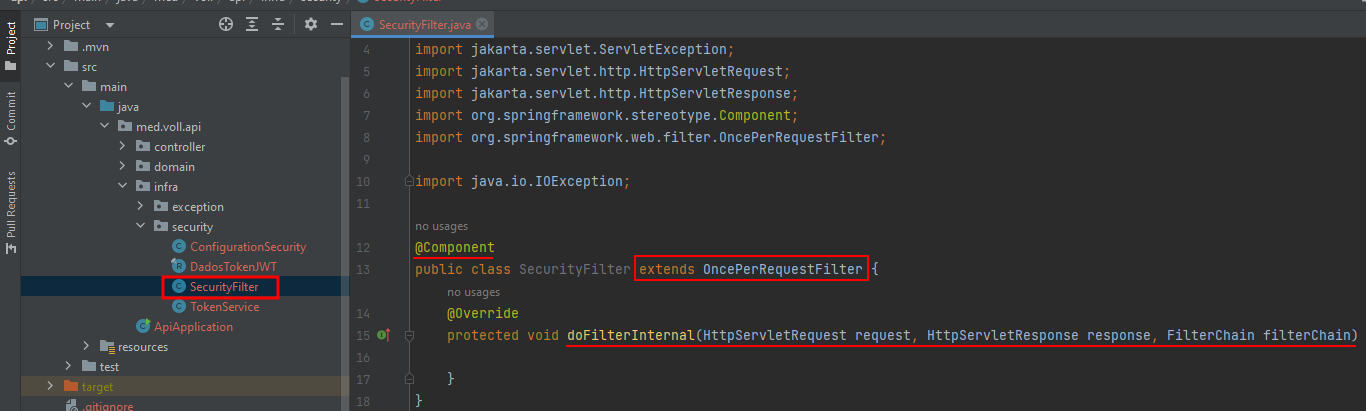
Em outras palavras, a requisição passará pelo filtro antes de cair no controller.

## Criando o filter de segurança

Para criar o nosso filtro iremos criar a classe SecurityFilter.

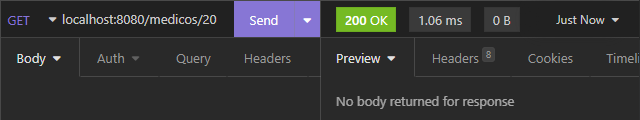
Nela iremos extender a OncePerRequestFilter, do próprio Spring, e implementar seu método doFilterInternal.

Como o Spring não conseguirá carregar a classe automaticamente no projeto, precisaremos passar a anotação @Component no código.



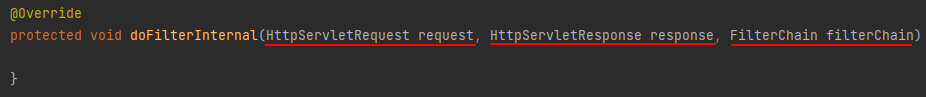
Com isso, TODAS as requisições irão antes passar por aqui.

Como em nosso filtro não especificamos qual próximo filtro chamar, no caso nossas requisições, ele devolvera tudo 200 sem nada.

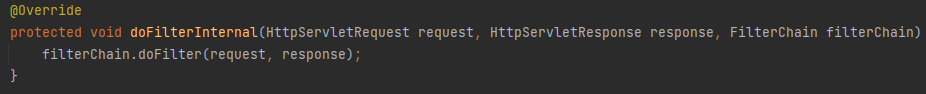


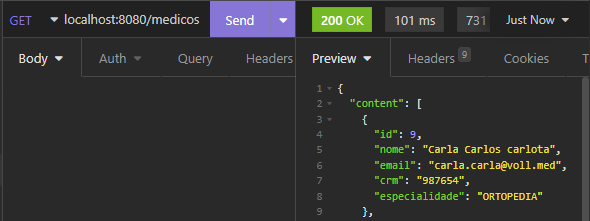
O método que implementamos recebe 3 parâmetros

* Reques: Usado para pegar informações da requisição
* Response: Usado para enviar informações na resposta
* filterChain: representa a cadeia de filtros na aplicação.



Para a nossa requisição dar continuidade é só utilizar o filterChain passando a request e response.

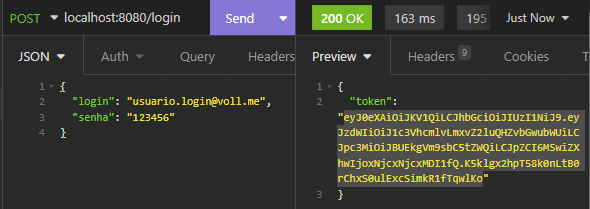




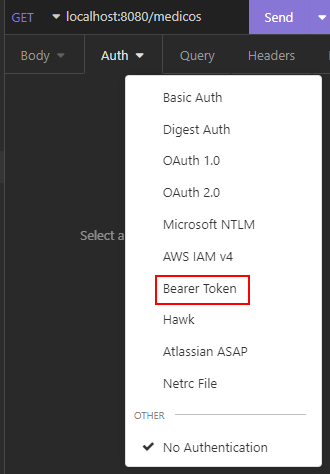
Com isso, se quisermos que nossa requição seja interrompida, basta criar uma lógica antes da linha do filterChain.

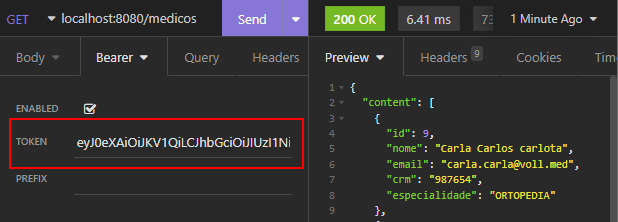
## Recuperando o token

Para passar o token em nossas requisições primeiro precisaremos logar, gerá-lo e copiá-lo:

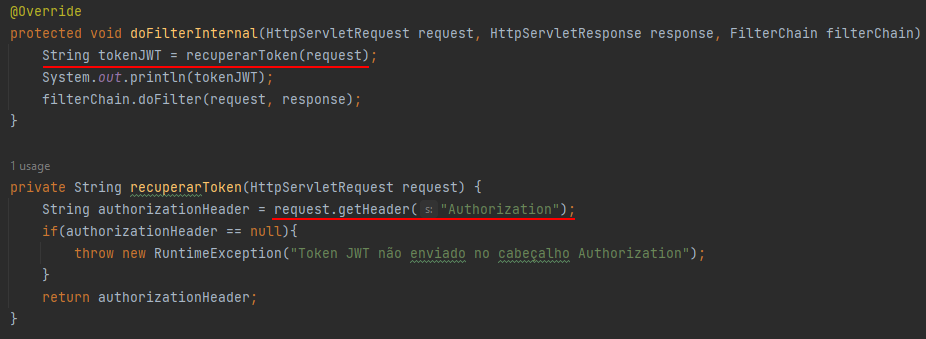


Logo em seguida, em nossa requisição, iremos passar o token conforme a imagem:

Só colar e solicitar a requisição



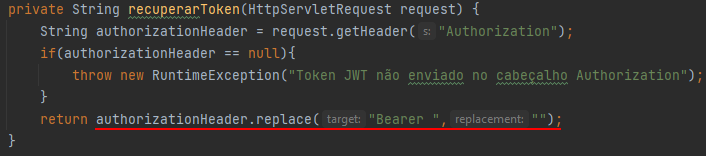
Em nosso filtro vamos pegar o header Authorization e resgatar o token



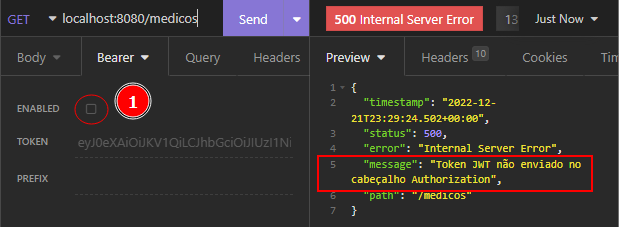


Vemos que foi impresso no console a palavra Bearer e junto o token. Esse Bearer é um prefixo que sinaliza o tipo de token, que por padrão os Bearer são para token JWT.

Vamos retirar esse “Bearer “, só para ficar mais fácil validar o token.



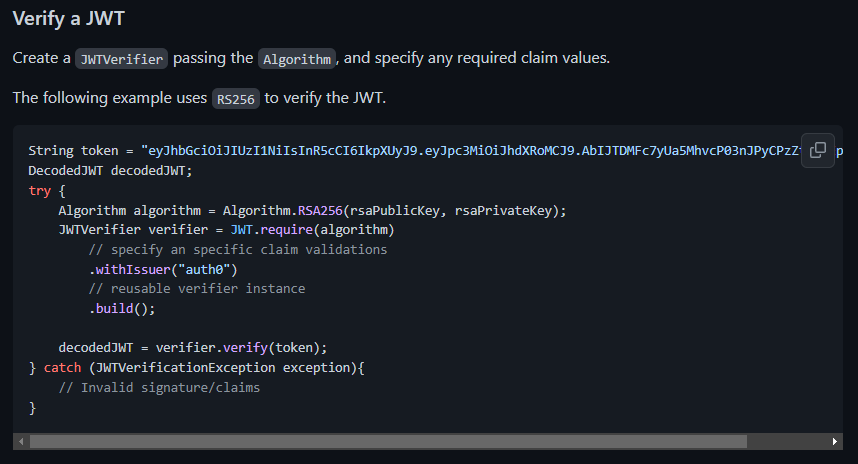




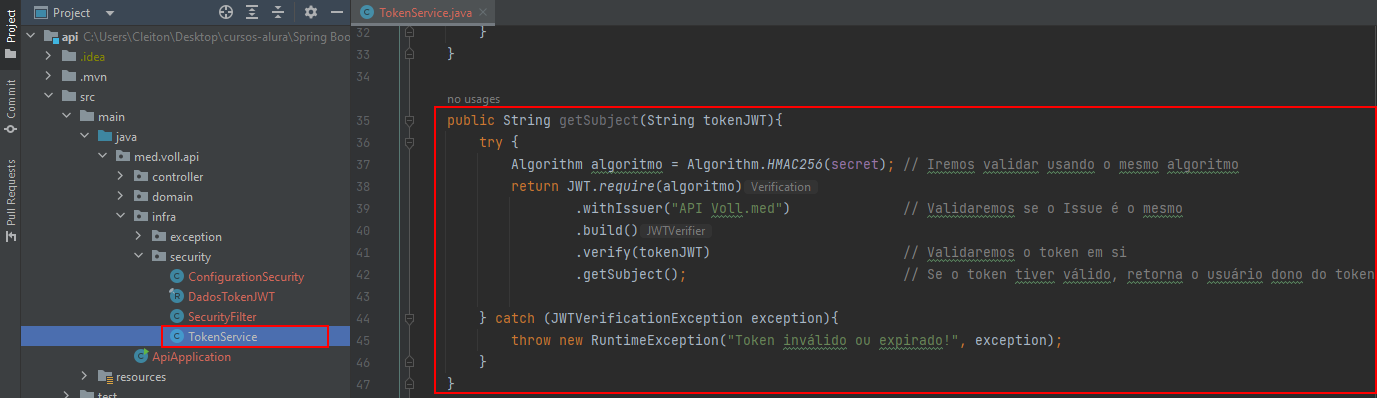
## Validando o token recebido

Na documentação da JWT, assim como pegamos o exemplo de como gerar um token também é demonstrado como valida-lo.

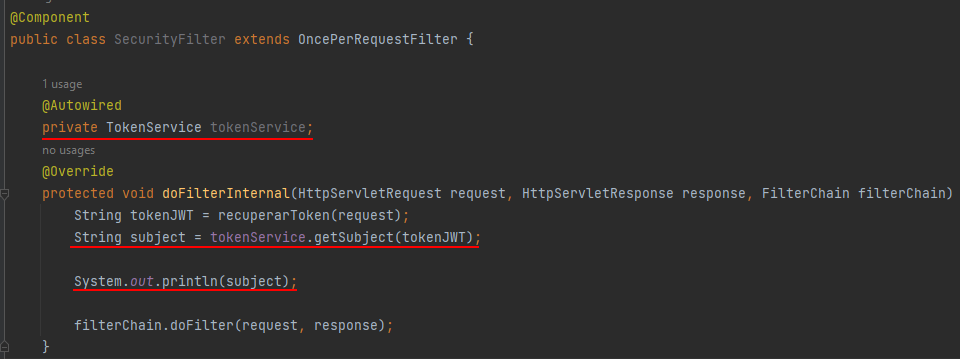
<https://github.com/auth0/java-jwt>

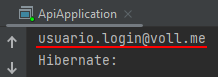


Vamos criar um método para validar o nosso token em TokenService



Implementar a validação em nosso filter:

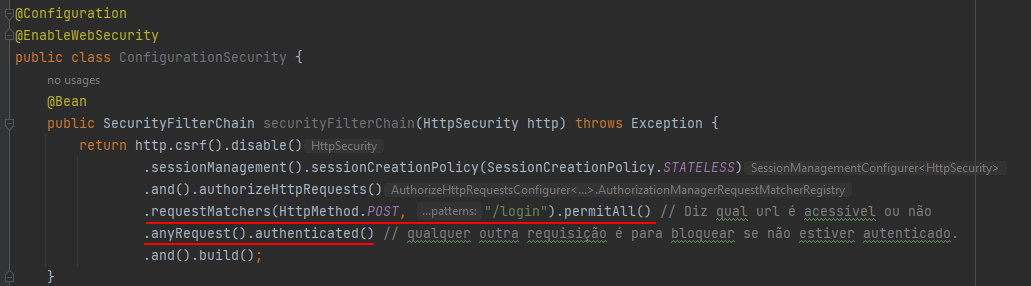




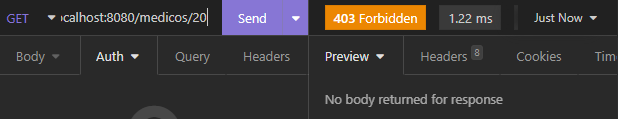
## Autenticando o usuário

Precisamos dizer para o Spring que o usuário está autenticado e que ele pode acessar as nossas requisições. Outra coisa é que todas as requisições devem ser bloqueados, menos o login.

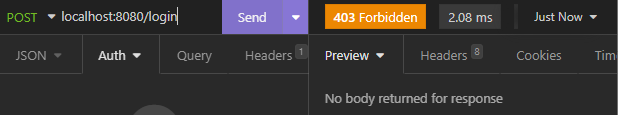
Alterando configurações de acessos em SecurityConfigurations:



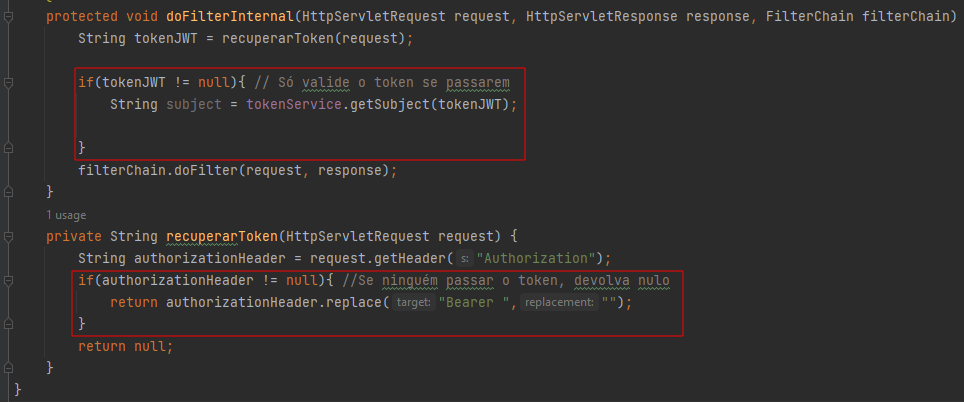
Testando:

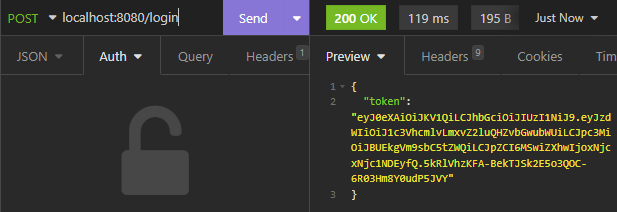


Login também foi afetado e não deveria:



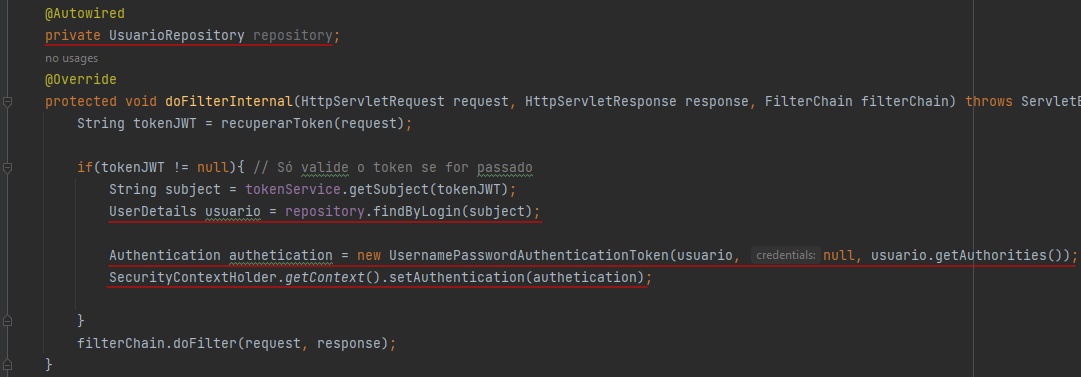
Ajustando filter:



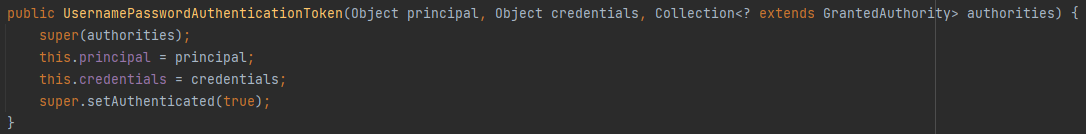


Forçando autenticação do usuário para o Spring:

Primeiro precisamos resgatar nosso usuário, depois criar um Authentication e forçar a autenticação.

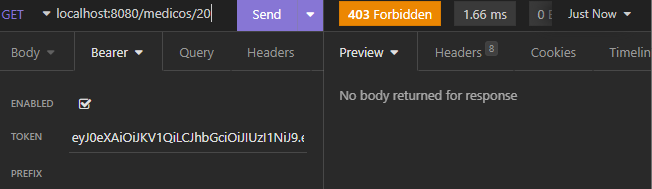


Vendo mais de perto a Classe UsernamePasswordAuthenticationToken

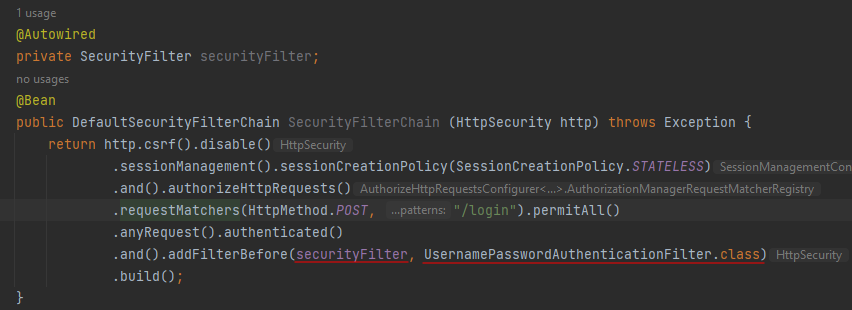


## Última configuração:

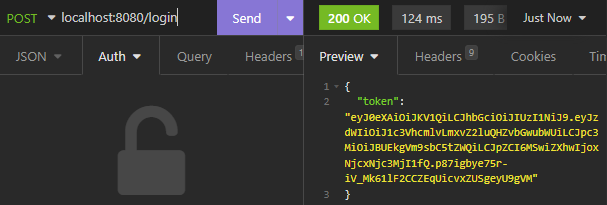
Mesmo após todas as configurações as nossas requisições continuam sendo bloqueadas

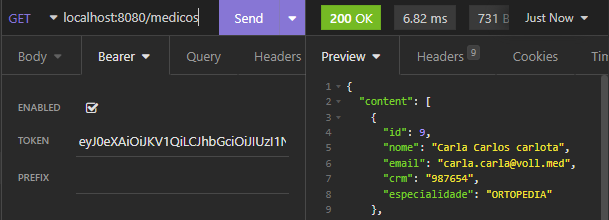


Isso está acontecendo pois no momento que mechemos nas requisições que podem ou não serem acessadas o Spring executa o seu filtro primeiro. Ao dar erro na requisição, o Spring nem passa para o próximo filtro, que seria o nosso. Para corrigir temos que falar ao Spring que precisa implementar nosso filtro antes.



No método addFilterBefore() passamos o filtro que iremos executar antes e qual será a seguinte, no caso, passamos a própria classe do Spring.





FIM!