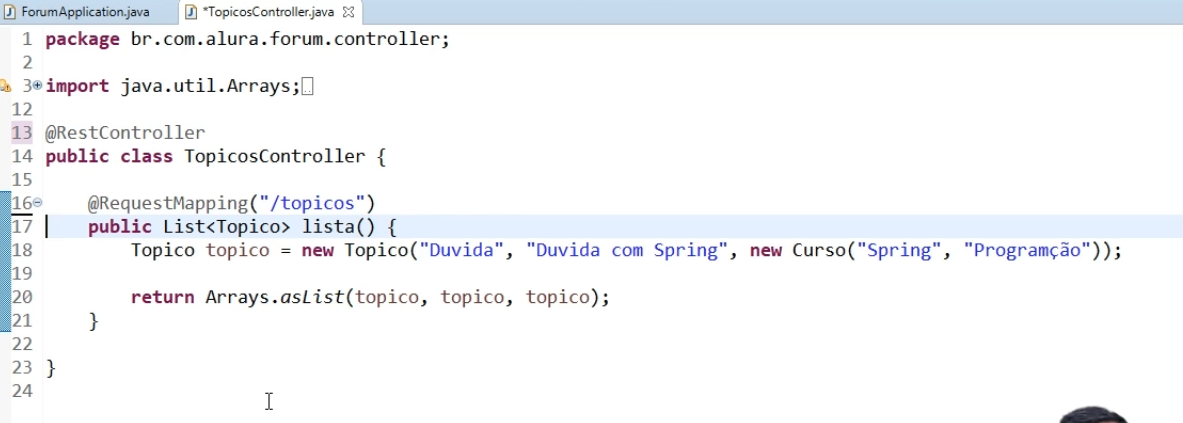
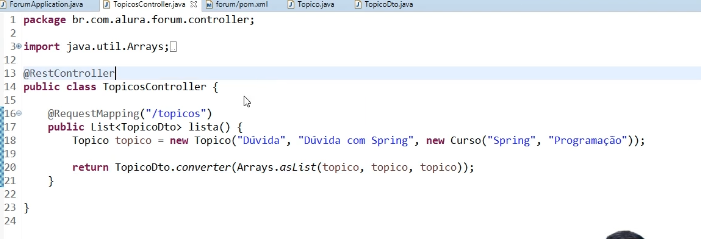
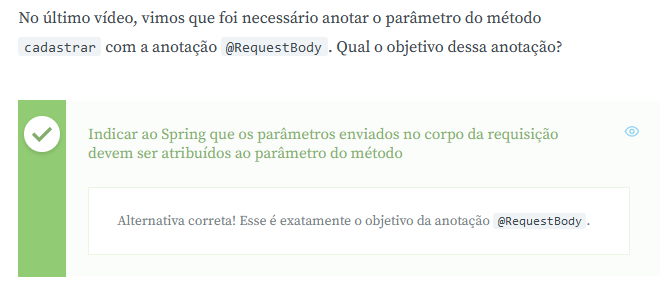
**Quando trabalhamos com Spring REST, ou seja, não quero retornar no navegador uma p6ágina HTML/JSP/Thymeleaf, quero retornar uma response body para uma API, eu utilizo o RestController ao invés do Controller somente:**

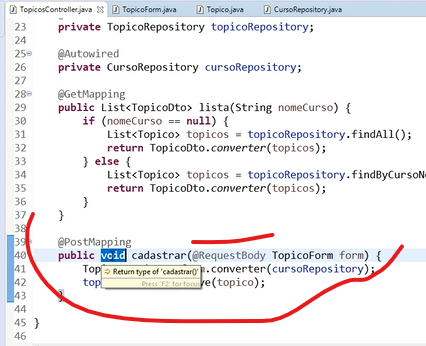


**Vale lembrar que não é legal devolver classes de domínio direto em um endpoint, temos que criar classes DTO ou VO que nos ajudam a pegar somente os campos do domínio que queremos pegar pra devolver na requisição, para que não tenhamos problemas de um domínio que tem outros domínios como dependência nos gerarem erros encadeados nestes domínios dentro de domínios, o que pode nos dar um problema de manutenção de código:**









**Para seguir as boas práticas do REST, deixamos nosso POSTMAPPING do controller mais certinho nos padrões:**

* **O retorno passou a ser do tipo ResponseEntity de TopicoDTO para que possamos passar na resposta da requisição o objeto tópico que foi criado (usando o DTO dele);**
* **Usamos o uriBuilder do tipo UriComponentsBuilder pois ele nos ajudará a criar uma URI de retorno quando um objeto requisitado com post for criado;**
* **Essa URI é montada com URIBuilder passando o path após a requisição ser feita com sucesso, que é tópicos/id sendo que o id pegamos do objeto tópico que acabou de ser criado;**
* **Por fim o return se dá com o Response Entity de created (código HTTP 201), mostrando no corpo da response o objeto tópico que foi criado.**

**Vale lembrar também que em uma requisição POST será enviado via JSON a requisição. A anotação RequestBody nos garante a ajuda para transformar o JSON recebido em um objeto do tipo que passamos ali no caso como TopicoForm, que é um objeto Java, usando o Jackson por baixo dos panos na conversão de JSON para Objeto JAVA.**

**Segue exemplo abaixo de uma classe de Controller da API REST que construímos com o Spring Boot para podermos ter uma consulta de um objeto específico passando seu id:**

****

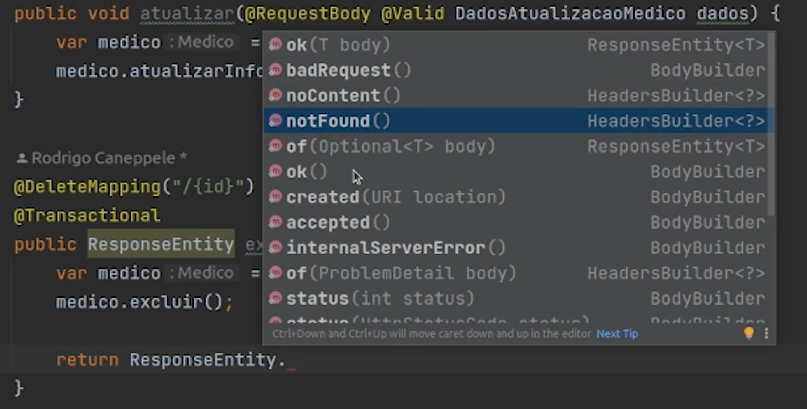
**Como é visto acima, a vantagem de usar DTOs é que com o objeto que pegamos do banco de dados eu só retorno na resposta da requisição o objeto no formato do DTO criado, seja com mais ou menos informações, sendo que cada ocasião podemos criar um DTO para ter um padrão e casca para cada retorno de um mesmo objeto que temos.**

****

**Com o uso do Optional conseguimos passar uma validação para verificar se o valor está presente dos ids e operações que queremos fazer. Se tiver um erro ou algo do tipo, é retornado um 404 com notFound:**

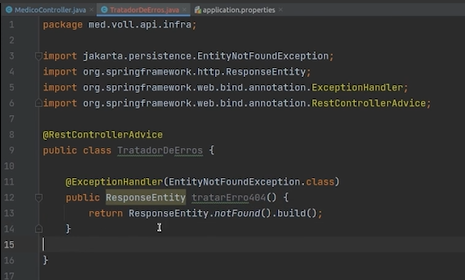
****

* **Boas práticas e protegendo uma API REST com Spring:**
* Sempre utilizar os retornos dos métodos do Spring para respostas HTTP para utilizar os códigos de status HTTP corretos, assim nunca retornando o método com void e sempre com Response Entity, aí podemos escolher os status HTTP com o Spring:

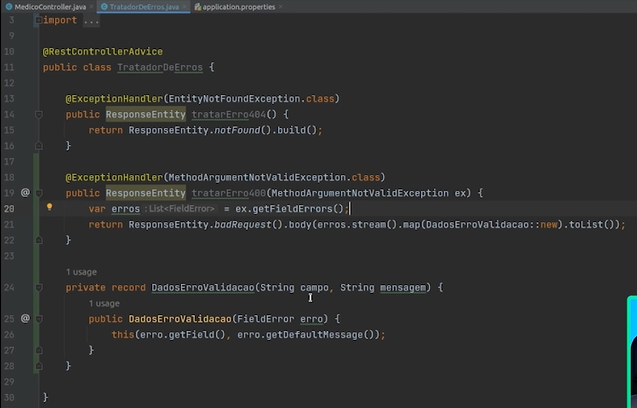


* **Tratamento de erros na API em suas requisições:**

Temos uma forma para conseguirmos tratar erros quando o usuário por exemplo tenta requisitar um registro não existente. Por padrão o Spring sempre retorna 500 para qualquer erro, mas nesse caso podemos configurar para retornar erro 404 de Not Found:



**Na imagem acima vemos que criamos uma classe para nos ajudar neste tratamento de erros, onde usamos a anotação RestControllerAdvice para isso passando um ExceptionHandler.**

****

**Acima temos um tratamento de erros para 400 onde pego a lista de erros da exceção e trabalho com ela dentro de um map para retornar a lista de erros no body da resposta pro usuário saber os campos e os erros de cada campo. Utilizei o record, que é um DTO para pegar somente a mensagem e o campo do erro para ser retornado.**

Você deve ter notado que o *Bean Validation* possui uma mensagem de erro para cada uma de suas anotações. Por exemplo, quando a validação falha em algum atributo anotado com @NotBlank, a mensagem de erro será: ***must not be blank***.

Essas mensagens de erro não foram definidas na aplicação, pois são mensagens de erro **padrão** do próprio *Bean Validation*. Entretanto, caso você queira, pode personalizar tais mensagens.

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

public record DadosCadastroMedico(

@NotBlank(message = "Nome é obrigatório")

String nome,

@NotBlank(message = "Email é obrigatório")

@Email(message = "Formato do email é inválido")

String email,

@NotBlank(message = "Telefone é obrigatório")

String telefone,

@NotBlank(message = "CRM é obrigatório")

@Pattern(regexp = "\\d{4,6}", message = "Formato do CRM é inválido")

String crm,

@NotNull(message = "Especialidade é obrigatória")

Especialidade especialidade,

@NotNull(message = "Dados do endereço são obrigatórios")

@Valid DadosEndereco endereco) {}

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:

nome.obrigatorio=Nome é obrigatório

email.obrigatorio=Email é obrigatório

email.invalido=Formato do email é inválido

telefone.obrigatorio=Telefone é obrigatório

crm.obrigatorio=CRM é obrigatório

crm.invalido=Formato do CRM é inválido

especialidade.obrigatoria=Especialidade é obrigatória

endereco.obrigatorio=Dados do endereço são obrigatórios

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

public record DadosCadastroMedico(

@NotBlank(message = "{nome.obrigatorio}")

String nome,

@NotBlank(message = "{email.obrigatorio}")

@Email(message = "{email.invalido}")

String email,

@NotBlank(message = "{telefone.obrigatorio}")

String telefone,

@NotBlank(message = "{crm.obrigatorio}")

@Pattern(regexp = "\\d{4,6}", message = "{crm.invalido}")

String crm,

@NotNull(message = "{especialidade.obrigatoria}")

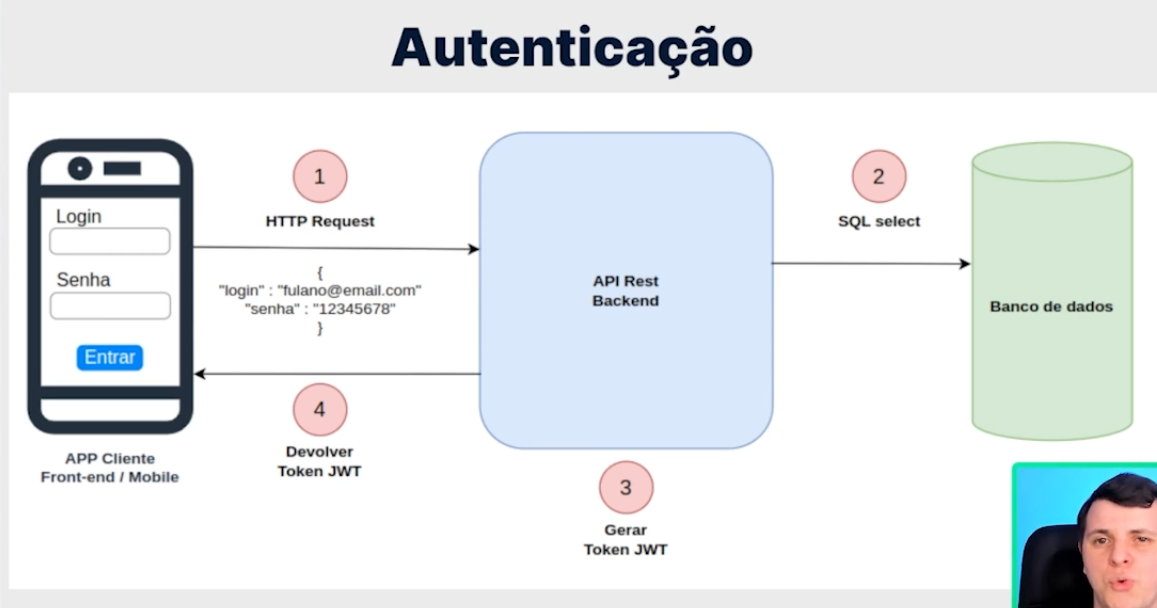
Especialidade especialidade,

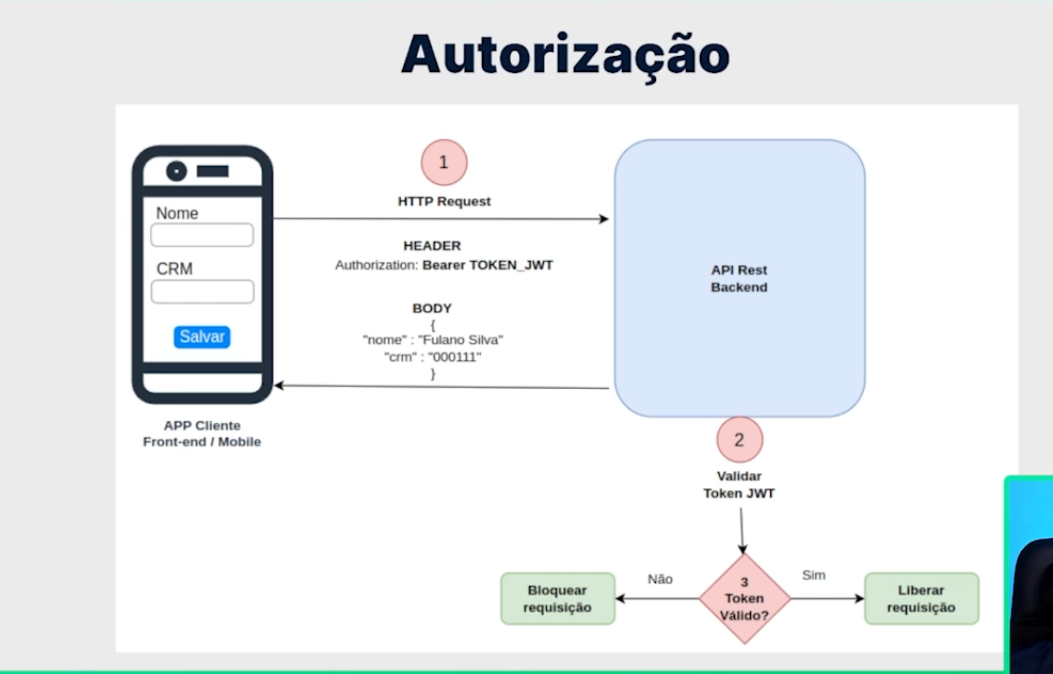
@NotNull(message = "{endereco.obrigatorio}")

@Valid DadosEndereco endereco) {}

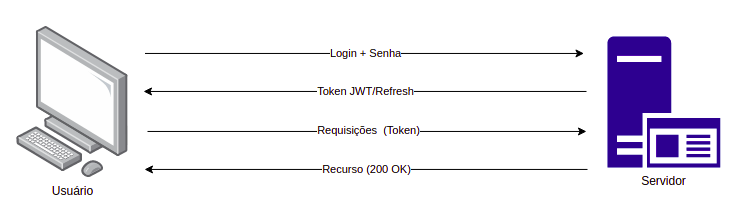
* **Utilizando o Spring Security para proteger a API REST Spring:**

**Para protegermos nossa API de requisições que possam ser públicas utilizamos tokens, onde o JWT – Json Web Token nos ajuda, para sempre termos um token solicitado pelas requisições para poder acessar os dados da API.**

****

****

**Vimos acima que após autenticar o usuário, a aplicação retorna um token ao usuário e ele só pode fazer requisições e ações na API da aplicação com o token passado válido no header das requisições.**



É importante salientar que informações do usuário não ficarão salvas no servidor. Elas são gravadas no token, que tem um período geralmente curto para expiração, por volta de 10 minutos. O tempo dependerá dos requisitos de segurança da aplicação.

Se um token chegar a expirar, uma solução seria pedir ao servidor um novo token válido. Isso acarretaria um novo processo de autenticação por login e senha ou por outro método escolhido. Porém, não é prático que a aplicação solicite usuário e senha novamente a cada expiração. Uma estratégia para lidar com essa situação seria usar *refresh token*.

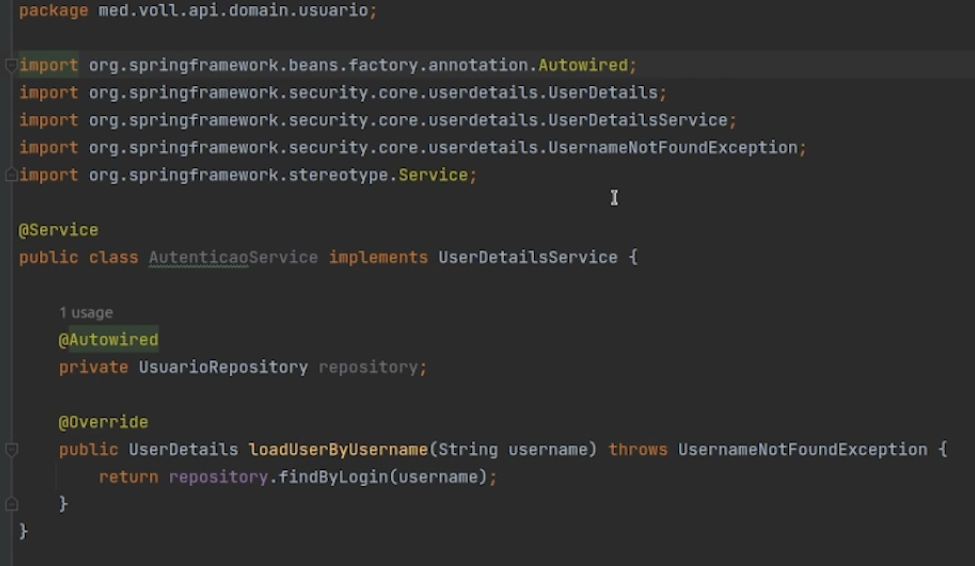
Como uma solução comum, costuma-se enviar tanto o token de acesso quanto o *refresh token*, assim que a autenticação na aplicação é realizada. No entanto, o *refresh token* possui um tempo maior para a expiração, e neste caso não armazenamos nenhuma informação de usuário.

Outra característica do *refresh token* é que ele é de uso único, então na próxima solicitação é enviado um token e um novo *refresh token*.

**Vamos lá começar a proteger a API:**

1. **Criar base de usuários e service para autenticação:**

**Quando trabalhamos com autenticação de usuários no Spring e queremos que tenhamos uma base para guardar as informações de login e senha dos usuários, antes de mais nada precisamos criar um repository para esta base e um service para ajudar no processo de autenticação, este service tem que extender à interface do Spring chamada UserDetailService, que serve para realizar o login do usuário e verificar se está de acordo com o que tem na base de usuários que criamos:**



**Texto

Descrição gerada automaticamente**

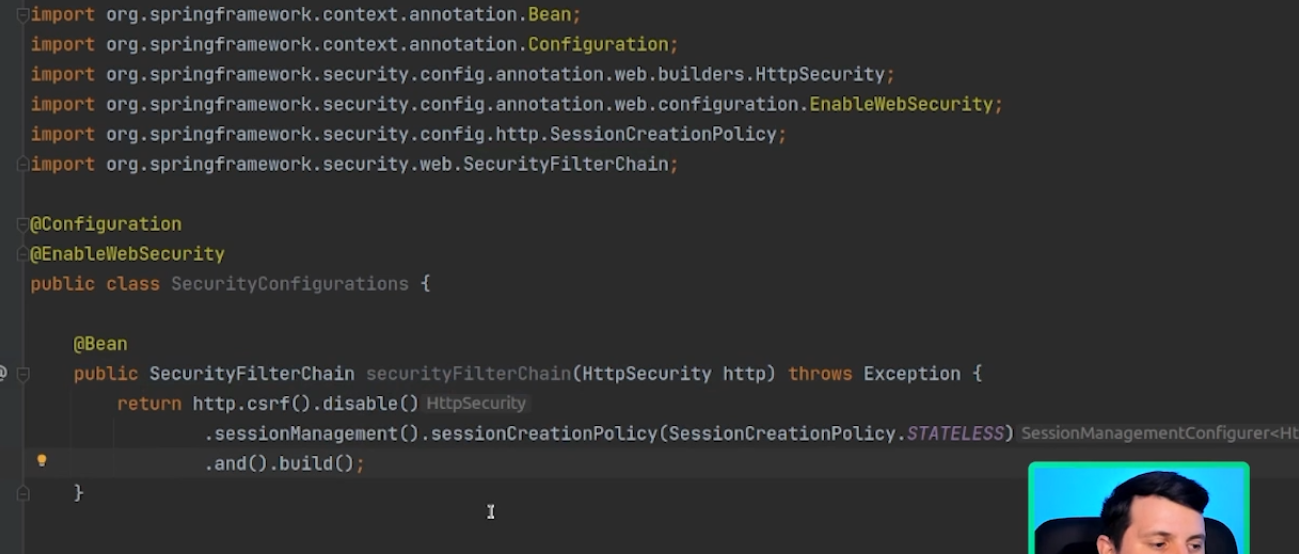
**No exemplo acima temos que utilizar o AuthenticationManager para realizar o trabalho de autenticação do login. Pegamos no corpo da requisição os dados do login passado pelo usuário, guardamos ele como se fosse um token, e aí passamos no manager.authenticate para autenticar este usuário.**

**Não esquecemos de passar para a classe de entidade a implementação de UserDetails para o Spring entender que é uma entidade para uso de autenticação:**

**Texto

Descrição gerada automaticamente**

1. **Configurações de segurança da API com SecurityConfigurations:**

****

**No caso acima habilitamos a autenticação stateless, ou seja, sem tela de login e front end, só para o meu backend com API mesmo. Quando tela de login e tudo no front é stateful.**

**Texto

Descrição gerada automaticamente**

**Não podemos esquecer de colocar no SecurityConfigurations também a instanciação do AuthenticationManager como um Bean para podermos utiliza-lo no controller de autenticação de usuários.**

**Também na classe SecurityConfiguration podemos passar para o Spring qual é o hash de senhas que está sendo utilizado para armazenar as senhas no banco de dados para ele conseguir saber para autenticar:**

**Texto

Descrição gerada automaticamente**

1. **Após a autenticação do usuário pegar o token da sessão e trabalhar com ele:**

**Após a requisição da autenticação der sucesso, no retorno temos que devolver não só o código 200, mas sim um token para que o usuário utilize durante sua sessão de autenticação. Podemos utilizar uma lib apartada para nos ajudar nessa geração de token no retorno da API:**

**Interface gráfica do usuário, Site

Descrição gerada automaticamente**

**Interface gráfica do usuário, Texto

Descrição gerada automaticamente**

**Mas para passar o token para esses retornos no Controller de Autenticação, a gente cria um Service que servirá essa criação de token:**

**Texto

Descrição gerada automaticamente**

**Agora no controller podemos usar o token gerado acima:**

**Texto

Descrição gerada automaticamente**