**Spring Boot API Rest: Segurança da API, Cache e Monitoramento**

Paginação

Imagine que o banco de dados começa a crescer e daqui a pouco tem milhares de registros. Toda vez que esse método lista for chamado, vamos devolver todas essas informações do banco de dados, e com muitos registros pode começar a ficar lento.

Para resolver esse problema, o pessoal usa a ideia de paginação, onde posso controlar. Ao invés de devolver todos os registros, posso devolver de pouco em pouco.

Vamos receber os parâmetros pela URI para indicar o comportamento da paginação, no geral, para esse tipo de parâmetro é interessante colocar uma anotação, que é o @RequestParam, para avisar ao Spring que é um parâmetro de request. Automaticamente, quando você coloca essa anotação, o Spring considera que o parâmetro é obrigatório. Se chamarmos esse endereço e não passarmos o parâmetro, o Spring vai jogar um erro 400 para o cliente dizendo que tem um parâmetro obrigatório que ele não enviou.

Só que no caso, o nome do curso vai ser opcional. Então, nessa anotação tem um atributo chamado required, que vai ser igual a falso.

Já o int página e int quantidade também vou colocar o request param, para avisar para o Spring que são parâmetros de url, mas vou colocar como obrigatórios.

Se você já trabalhou com JPA, ela tem suporte para paginação. Se você usa aquela API do entitymanager, na hora de criar sua query tem métodos para setar o maxResult, firstResult, e aí você consegue controlar qual o primeiro registro e qual o número máximo de registros. Dessa maneira, você faria a paginação com JPA na mão. Mas aqui estamos usando o Spring data. Lembre-se que ele abstrai algumas coisinhas para nós, inclusive paginação.

 Se quisermos fazer paginação, o Spring data já tem um esquema pronto para facilitar nossa vida. Para fazer paginação, precisamos criar uma variável do tipo pageable, que é uma classe do Spring data que serve justamente para fazer paginação. Vou criar essa variável, vou chamar de paginação = page.

Para criar essa interface pageable, vou precisar importar. Mas cuidado, tem três opções. A que queremos é a do org.springframework.data. Para criar esse cara, usamos outra classe chamada pageRequest. Nela, tem um método estático chamado of, em que passamos a página e a quantidade. Com isso, ele cria um objeto do tipo pageable.

Todo ponto que recebia um list numa busca get no caso a que estamos mudando irá retornar uma Page.

Como vimos no vídeo anterior, foi necessário alterar o retorno do método lista, de List<Topico>, para Page<Topico>. Qual a vantagem de devolver um objeto Page, ao invés de um List?

* Adicionar dados sobre a paginação no JSON de resposta, ao utilizar o objeto Page, além de devolver os registros, o Spring também devolve informações sobre a paginação no JSON de resposta, como número total de registros e páginas.

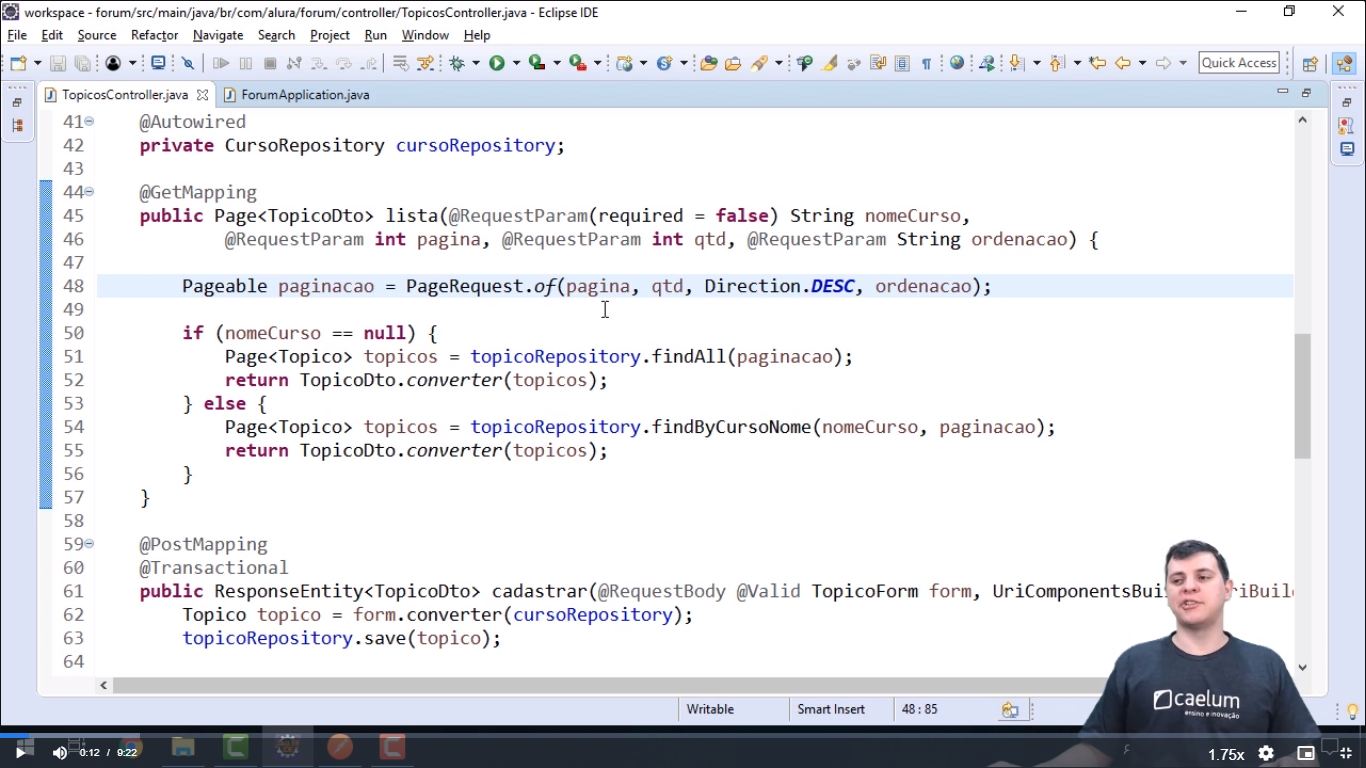
Ordenação

Vamos flexibilizar, deixar que o cliente consiga controlar também qual ordem ele quer, por qual atributo do tópico ele quer ordenar os registros.

Para fazer isso, mais um parâmetro que podemos colocar no nosso método. No caso, vou criar um parâmetro chamado ordenação, que vai ser uma string. A ideia vai ser que além de passar a página e a quantidade de registros, o cliente também vai ter que passar qual o campo que ele quer ordenar. A ideia dessa string é passar o nome do campo, que seria o atributo da nossa classe tópico. Pode ser id, data, status ou qualquer outro atributo da nossa classe tópico.

Vimos que, para ordenar os registros, foi necessário passar um parâmetro com o nome do atributo para realizar a ordenação. O que acontece se o nome do atributo informado estiver incorreto?

Ao informar um atributo que não existe, um erro 500 será devolvido.



Uma maneira mais fácil de realizar a paginação é que apagamos essa linha, e ao invés de receber os três parâmetros soltos, a página, quantidade e ordenação, já conseguimos receber todos juntos em um objeto pageable. Aquele objeto pageable conseguimos receber como parâmetro direto do nosso método lista. Não preciso receber os três campos soltos. Posso receber o pageable, e dentro dele já vem todas as informações de paginação e ordenação. A ideia é deixar o método com menos parâmetros e tirar aquela linha que estava criando o pageable na mão.

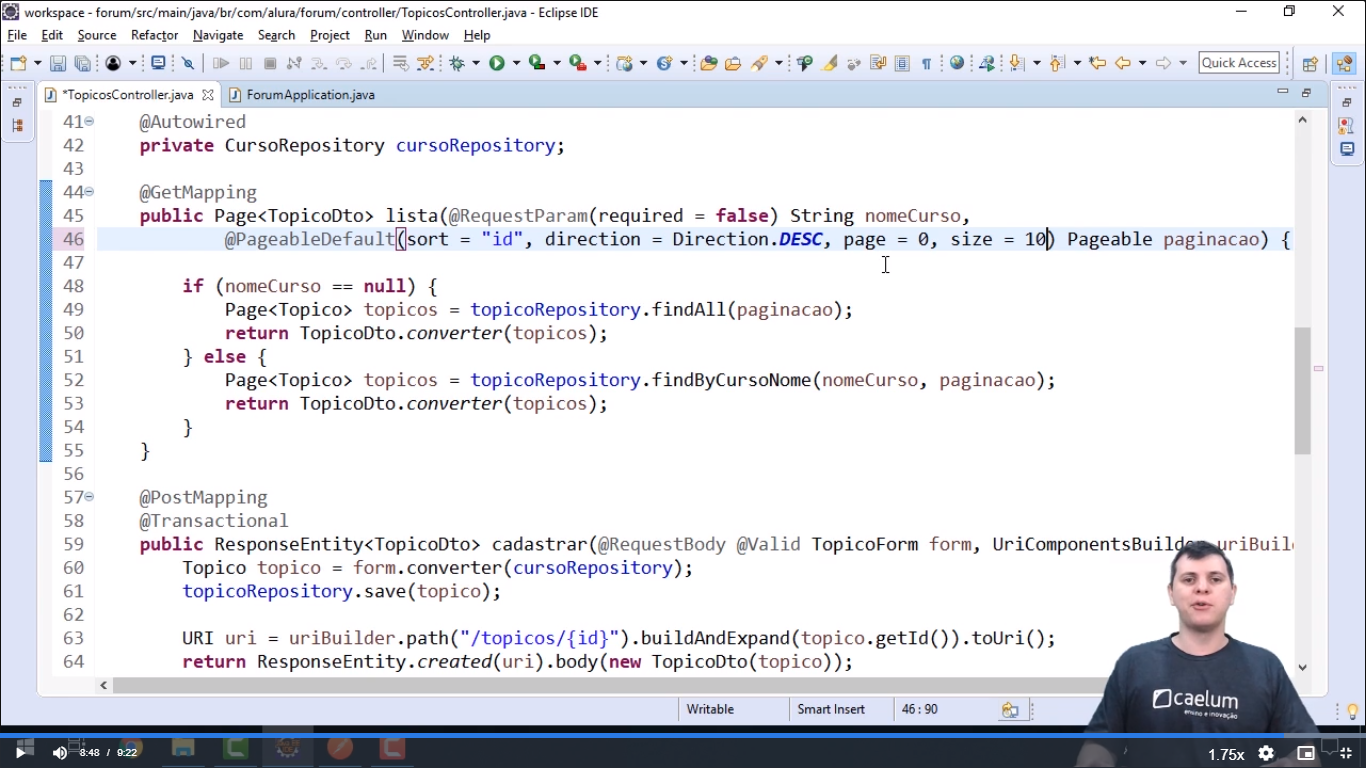
Só que tem um problema. Para o Spring conseguir pegar os parâmetros de paginação e ordenação da requisição, ele precisa que o módulo esteja habilitado no projeto, que é o módulo que faz esse suporte de pegar as coisas da web e passar para o Spring data. Esse módulo não vem habilitado no projeto por padrão. Para habilitarmos isso, quase todas as coisas que alguns módulos que vamos utilizar do Spring Boot, que não vem habilitado por padrão, temos que ir até aquela classe fórum application, que é nossa classe main.

Em cima da classe, só tinha o **@SpringBootApplication**, agora temos que colocar uma nova anotação, que é o **@enableSpringDataWebSupport**. Com essa anotação habilitamos esse suporte, para o Spring pegar da requisição, dos parâmetros da url os campos, as informações de paginação e ordenação, e repassar isso para o Spring data.

Porém, estávamos passando em português. Para o Spring pegar automaticamente, ele precisa que os parâmetros estejam em inglês, com determinado nome. O nome da página tem que ser page=0&size=10&sort=id,asc. Você separa com vírgula e coloca asc para crescente e desc para decrescente. É bem simples.

Você também consegue dizer qual a ordem padrão.

No nosso código, no parâmetro pageable, existe uma anotação que podemos colocar chamada @PageableDefault. Nesse PageableDefault tem alguns parâmetros. Consigo dizer que sort = id, Direction.DESC. Com isso, ele está dizendo que a paginação default é id de maneira decrescente. O default é: se não estiver vindo um parâmetro de ordenação.



Qual o objetivo da anotação @PageableDefault?

Indicar o padrão de paginação/ordenação ao Spring, quando o cliente da API não enviar tais informações

Nesta aula, aprendemos que:

* Para realizar paginação com Spring Data JPA, devemos utilizar a interface Pageable;
* Nas classes Repository, os métodos que recebem um pageable como parâmetro retornam objetos do tipo Page<>, ao invés de List<>;
* Para o Spring incluir informações sobre a paginação no JSON de resposta enviado ao cliente da API, devemos alterar o retorno do método do *controller* de List<> para Page<>;
* Para fazer a ordenação na consulta ao banco de dados, devemos utilizar também a interface Pageable, passando como parâmetro a direção da ordenação, utilizando a classe Direction, e o nome do atributo para ordenar;
* Para receber os parâmetros de ordenação e paginação diretamente nos métodos do *controller*, devemos habilitar o módulo SpringDataWebSupport, adicionando a anotação @EnableSpringDataWebSupport na classe ForumApplication.

**Utilizando cache**

 A ideia é que se eu utilizar o cache, consigo dizer para o Spring guardar o retorno de um método em cache. Na primeira vez que eu chamar aquele método, ele vai executar linha por linha do método, vai devolver o resultado, mas nas próximas chamadas ele já devolve direto o que está em memória. Com isso, fica muito mais rápido o retorno para o cliente que fez a chamada para a API.

O primeiro passo é: precisamos adicionar esse módulo do Spring Boot, que é um modelo de cache no nosso projeto. Não temos ele adicionado como dependência do Maven. Vamos abrir o .xml, e posso pegar alguma das dependências, por exemplo, da JPA, e só trocar o artifactid, para Spring Boot starter cache. Salvei. Ele vai baixar as dependências do Spring Boot starter cache, colocar no nosso projeto.

Outra dependência que deveríamos colocar é de qual a ferramenta, o provedor de cache que quero utilizar para aplicação. Com a aplicação rodando em produção, o ideal é utilizar algum provedor de cache. O Spring suporte alguns provedores, mas para fins de desenvolvimento, não precisamos usar nenhum provedor. Se não declararmos um provedor, o Spring usa um padrão, que é onde ele guarda o cache em memória usando um REST map. Mas esse provider não é recomendado para usar em produção.

Coloquei só a dependência do cache. Ele vai usar o provedor padrão, usando um REST map. O próximo passo é: além de declarar dependência de cache no projeto, precisamos habilitar o uso de cache na aplicação, porque ele não vem habilitado por padrão. Nós vimos no outro exemplo da paginação que para habilitar coisas no projeto temos que ir à classe fórum application e colocar uma anotação em cima dela.

Já temos o enable Spring data web support. Agora vamos utilizar outra aplicação, que é o @EnableCaching, para habilitar o uso de cache na aplicação. Vou salvar. Com isso, tenho o recurso de cache habilitado no projeto. Agora é só utilizar. Para utilizar cache, tenho que ir em alguma classe, controller, service ou repository, e em cima do método falar para o Spring que eu quero que ele guarde o retorno desse método em cache.

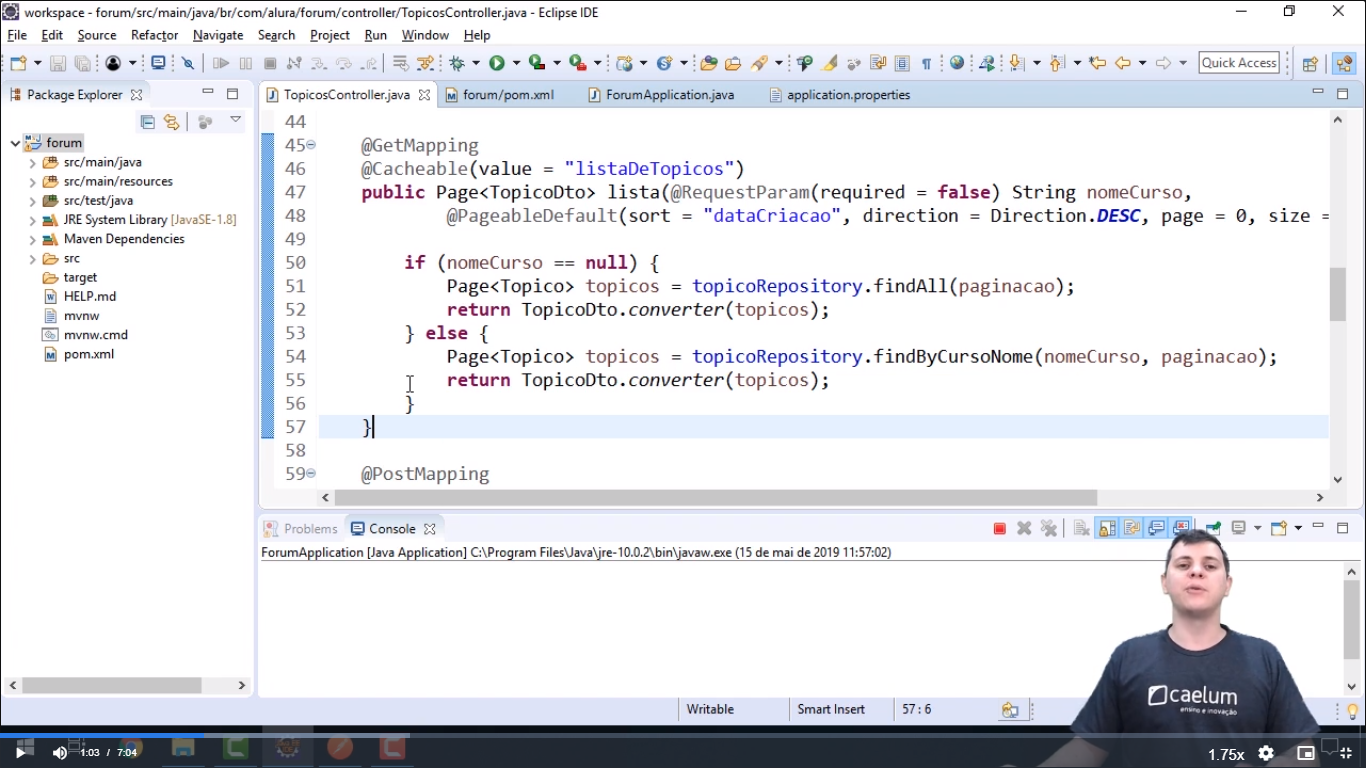
No nosso exemplo, vou colocar na nossa classe tópicos Controller um método lista em cache. Em cima do método, temos que colocar a anotação @Cacheable, para falar para o Spring guardar o retorno desse método em cache. Só cuidado na hora de fazer o import, porque existe a mesma anotação no pacote da JPA. O que vamos utilizar no curso é do org.springframework.

Essa anotação tem um atributo que precisamos preencher. Um chamado value, em que temos que passar uma string que vai ser o identificador único desse cache. Na nossa aplicação, posso ter vários métodos anotados com @Cacheable, e o Spring precisa saber como ele vai diferenciar um do outro. Ele faz isso utilizando o id único. Vou passar um nome, por exemplo, listaDeTopicos. Essa string vai funcionar como sendo um id desse cache.

No último vídeo, vimos que a anotação @Cacheable possui o parâmetro value, do tipo String. Para que serve esse parâmetro?

Para indicar ao Spring o nome do cache associado a um determinado método

A string passada como parâmetro para a anotação @Cacheable funciona como um identificador único do cache.



Se alguém cadastrar, excluir ou alterar um tópico no projeto, o cache não vai ser atualizado automaticamente. Na próxima consulta para listagem eu teria uma informação desatualizada. Só viria os resultados daqueles três caches que tenho cadastrados no banco de dados, mas o novo tópico que foi cadastrado, excluído ou alterado eu ainda não teria a informação, porque o cache estaria lá intacto.

Precisamos avisar para o Spring que quando um novo tópico for cadastrado, excluído ou alterado, ele precisa atualizar o cache, porque senão o usuário vai continuar recebendo uma informação desatualizada. Isso pode ser um problema.

Para fazer isso também é bem tranquilo. Tenho o tópico controller aberto. Vou procurar meu método salvar. Aqui, faço uma alteração nos tópicos. Quando esse método for chamado, quero que o Spring limpe aquele cache que chamei de lista de tópicos. Para dizer isso para o Spring é bem fácil. Em cima do método você coloca a anotação @CacheEvict, para dizer que quero que o Spring limpe determinado cache.

Mas como posso ter vários caches no projeto, preciso dizer para ele qual dos caches ele precisa limpar. Por isso, quando colocamos o @Cacheable, passamos no value um lista de tópicos. Na hora de fazer o @CacheEvict, preciso dizer no value qual dos caches quero limpar. Quero limpar o que eu chamei de lista de tópicos.

Tem outro parâmetro para limpar todos os registros. No nosso caso sim, quero limpar todos os registros, para ele atualizar tudo e deixar o cache zerado de novo. Vou salvar. E aí vamos fazer um teste no Postman.

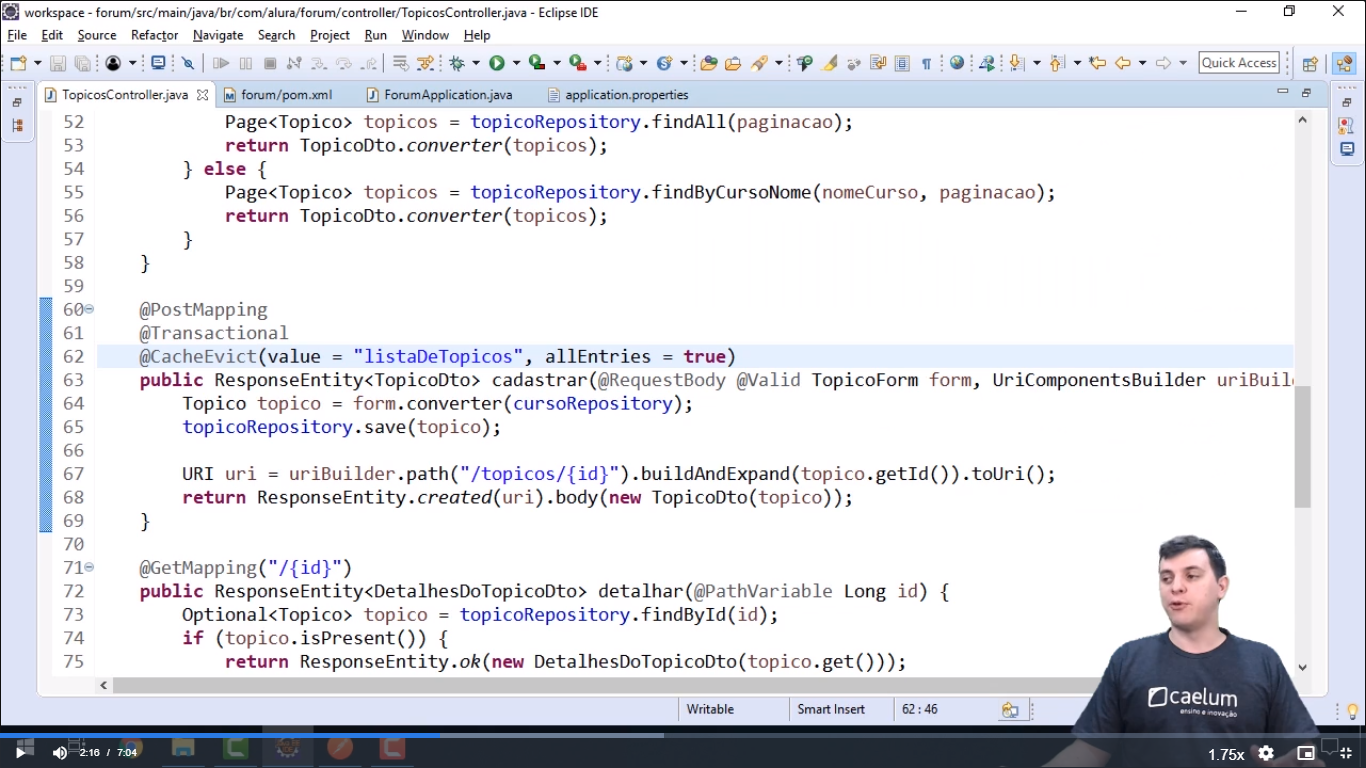
Sobre a utilização da anotação @CacheEvict, é correto afirmar que:

Ela deveria ser utilizada nos métodos que alteraram os registros armazenados em cache pela API

Isso é importante para evitar que os clientes obtenham informações desatualizadas.

Devemos indicar, no parâmetro value, quais caches devem ser invalidados

Nesse parâmetro devemos indicar os identificadores dos caches que deverão ser invalidados.



**Boas práticas no uso de cache**

Tomem cuidado no projeto de vocês, onde vão utilizar o cache. Procurem pensar um pouco antes de utilizar e utilizem nesse tipo de situação, nesses métodos que nunca ou raramente vão ser atualizados, porque assim você evita esse custo de limpar o cache e guardar de novo.

Você vai ter que pensar na sua aplicação. Sabe aquelas tabelas que nunca ou raramente são atualizadas? Quase todo projeto tem aquelas tabelas, por exemplo, de país, de Estado, de cidade, de tipos. É uma tabela estável. Esse é o tópico de entidade de consulta em que faz sentido utilizar cache, porque vou fazer a consulta uma única vez, vou guardar o resultado em memória, e aquele resultado vai ficar um bom período de tempo em memória. Ele não vai ser atualizado o tempo inteiro.

Nesta aula, aprendemos que:

* Para utilizar o módulo de cache do *Spring Boot*, devemos adicioná-lo como dependência do projeto no arquivo **pom.xml**;
* Para habilitar o uso de caches na aplicação, devemos adicionar a anotação @EnableCaching na classe ForumApplication;
* Para que o Spring guarde o retorno de um método no cache, devemos anotá-lo com @Cacheable;
* Para o Spring invalidar algum cache após um determinado método ser chamado, devemos anotá-lo com @CacheEvict;
* Devemos utilizar cache apenas para as informações que nunca ou raramente são atualizadas no banco de dados.

**Habilitando o Spring Security**

O Spring Boot possui um módulo focado somente nessa parte de segurança, que cuida da parte de autenticação e autorização, que é o Spring security. O primeiro passo para começarmos a proteger nossa API é baixar o módulo do Spring security e adicionar como uma dependência do nosso projeto.

**Spring-Boot-starter-security**, que é o módulo de segurança do Spring Boot. Ele vai baixar as dependências.

vou criar uma classe chamada, por exemplo, securityConfigurations. A ideia é que dentro dessa classe estarão todas as configurações de segurança do nosso projeto.

É uma classe Java, não tem nada a ver com Spring. Tenho que habilitar a parte do Spring security. Para fazer isso, fazemos na própria classe. Existe uma anotação chamada @EnableWebSecurity. Como essa é uma classe que tem configurações, precisamos colocar a anotação @Configuration. O Spring vai carregar e ler as configurações que estiverem dentro dessa classe.

Além disso, vamos ter que herdar essa classe de outra classe do Spring chamada web security configurer adapter. Essa classe tem alguns métodos para fazer as configurações que vamos sobrescrever posteriormente.

É isso. Nós colocamos a dependência do Spring security no projeto, criamos a classe, anotada com @EnableWebSecurity, com @Configuration. Dentro, depois, vamos colocar as configurações de segurança. Por enquanto está vazio, mas só de ter feiro isso já habilitamos a parte de segurança. Por padrão, o Spring bloqueia todo acesso à nossa API. Tudo está restrito até que eu faça a configuração e libere o que precisa ser liberado.

**Liberando acesso aos endpoints públicos**

Tentamos acessar a url /tópicos e recebemos como resposta o código 401, a ideia é configurar quais endereços quero liberar e quais quero proteger. Vou exigir que o cliente que está disparando a requisição esteja autenticado no sistema.

Para fazer isso, todas as configurações de segurança vão ficar na classe que criamos na última aula. Lembra que tivemos que herdar a classe de WebSecurityConfigurerAdapter? Nessa classe que estamos herdando existem alguns métodos onde as configurações de autenticação e autorização têm que ficar. Temos que fazer então uma sobrescrita, porque por padrão ele tem um comportamento específico, mas quero sobrescrever com as regras da minha aplicação.

Qual método tenho que sobrescrever? O método se chama configure. Mas tem um problema. Existem três métodos com esse nome. Nós vamos precisar usar os três. Vou escolher o primeiro, dar um enter. Pegar o segundo. E pegar o terceiro.

Qual é a diferença entre eles? O primeiro, que recebe um authentication manager builder é um método que serve para configurar a parte de autenticação. A parte de controle de acesso, de login, fica nesse método.

O segundo, que recebe um tal de http security, serve para fazer configurações de autorização. A parte de URLs, quem pode acessar cada url, perfil de acesso. E o terceiro, que recebe um tal de web security, serve para fazermos configurações de recursos estáticos. São requisições para arquivo CSS, Javascript, imagens, etc.

São esses três métodos configure que precisamos ter. Nesta aula, vamos utilizar o segundo. É nele que faço a configuração de autorização, das URLs do meu projeto, o que é público e o que preciso ter controle de acesso.

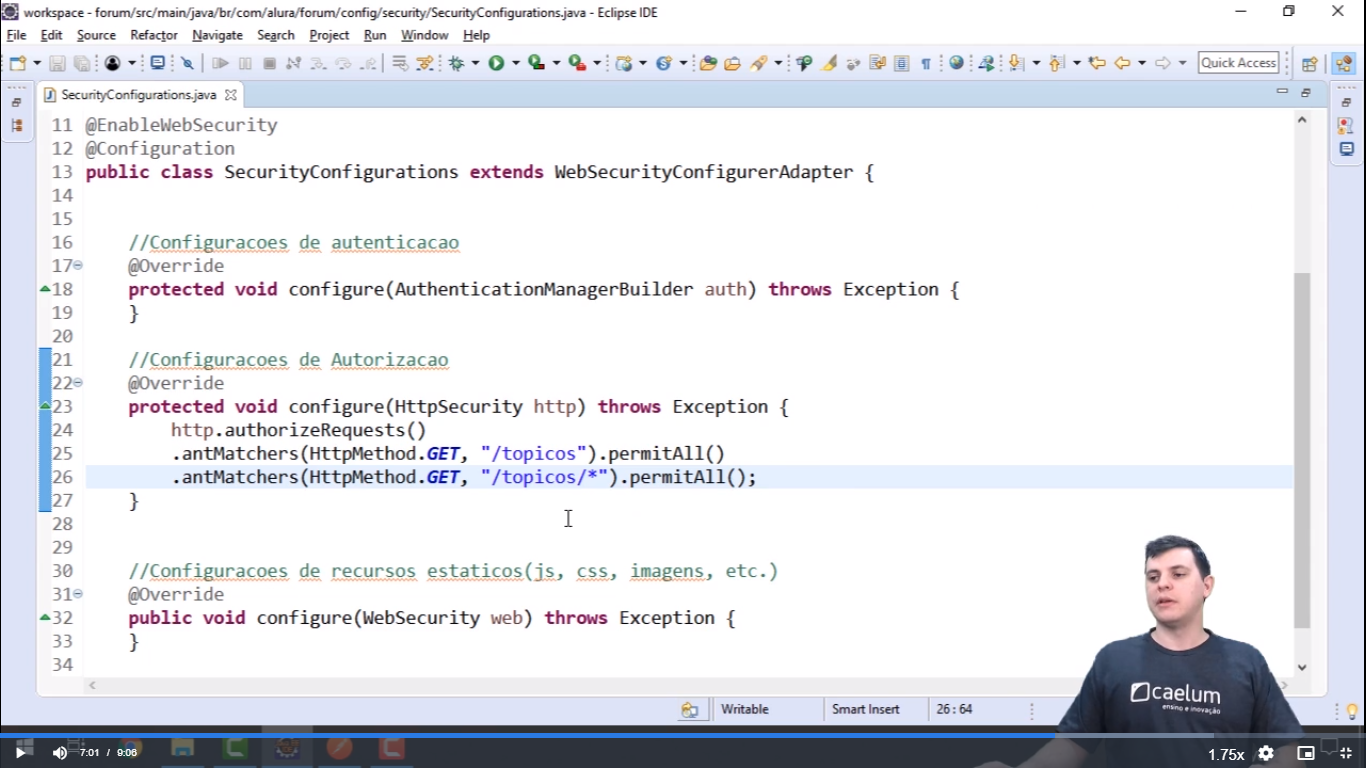
Percebe que esse método recebe um tal de http (da classe HttpSecurity)? E esse parâmetro tem alguns métodos, por exemplo, http.authorizeRequests é o método que vamos precisar chamar para configurar quais requests vamos autorizar, e como vai ser essa autorização.

Tem vários métodos, mas o que interessa para nós é o tal de antMatchers. Nós vamos falar para ele qual url quero filtrar e o que é para fazer, se é para emitir ou bloquear. Tem três versões dele. Na primeira, você pode passar só qual é a url, na sequência você diz se quer permitir ou não.

No nosso projeto, como vai ficar essa parte? Na nossa API quero deixar público o endpoint que lista todos os tópicos, ou seja, o /tópicos, e o que detalha um tópico em específico, o /tópicos/id. Os outros três, para cadastrar, alterar e excluir, quero restringir. Não é para ser público.

Do jeito que eu fiz, ele está liberando /tópicos independente do método. Eu quero liberar o /tópicos, mas não todos os métodos, só o método GET, que é para o método de listagem. Existe outra sobrecarga desse método antMatchers que antes de passar a url, posso passar que método quero filtrar.

Estou só liberando acesso aos meus endpoints públicos. A princípio, está ok, já fiz minhas configurações. Vamos testar então no Postman. Eu tinha tentando acessar o /tópicos e tinha dado 401. Agora, na teoria, é para liberar o acesso.



Vimos no vídeo anterior que, para liberar acesso ao endpoint de lista de tópicos, foi necessário adicionar a seguinte linha de código:

antMatchers(HttpMethod.GET, “/topicos”).permitAll();

Por que foi necessário indicar o método GET nessa configuração?

Para liberar acesso apenas às requisições do tipo GET

Se não indicarmos o método HTTP, todos seriam permitidos.

**Restringindo o acesso aos endpoints privados**

Na sequência, o ideal é colocar isso também: anyRequest().authenticated(). Qualquer outra requisição tem que estar autenticada. O Spring vai devolver 403, e só vamos conseguir acessar, disparar requisições para esse endereço se o cliente que estiver disparando as requisições estiver autenticado.

Para o cliente disparar requisições, ele vai precisar estar autenticado, mas aí precisamos ter a ideia de autenticação no projeto, vamos ter que ter, por exemplo, uma classe que representa o usuário, a senha dele, e o perfil de acesso.

orém, o Spring security precisa saber que essa é a classe que representa o usuário. Precisamos dizer para ele. Como ensinamos isso?

Vamos precisar implementar uma interface. Existe uma interface no Spring security que precisamos implementar na classe usuário. Preços implementar uma interface chamada UserDetails, que é a interface para dizer que essa é a classe que tem detalhes de um usuário. Dá erro de compilação porque no Java se você implementou uma interface, você é obrigado a sobrescrever, implementar os métodos que estão definidos nela.

São vários métodos que ele gerou, que precisamos sobrescrever. Por exemplo, o método Getpassword. Para saber a senha do seu usuário, o Spring vai chamar o método Getpassword, então precisamos devolver qual atributo representa a senha. No nosso caso, é this.senha.

Tem alguns métodos que desenvolvem boolean que é caso você faça controle na sua aplicação, da conta do usuário, se a conta está bloqueada, se tem data de expiração ou coisas do gênero, você devolveria os atributos que representam essas informações. No nosso caso não vamos ter esse controle mais fino, mais detalhado. Vamos devolver true em todos os métodos.

Faltou o método que tem que devolver uma collection de grantedAuthority. Para o Spring security, além de ter uma classe usuário, precisa ter uma classe também que representa o perfil do usuário. Qual o perfil relacionado com as permissões de acesso dele. Por isso ele tem mais esse método, que é para devolver qual atributo contém a coleção com os perfis desse usuário.

Na classe usuário, além do id, nome, e-mail e senha vamos ter que ter mais um atributo private. Vou dar um new só para inicializar a coleção, para não ficar nula, vou importar o ArrayList.

Nós não temos no nosso projeto uma classe perfil. Vamos criar essa classe, que vai ser uma entidade da JPA. Vou ter que ter uma tabela para guardar os perfis de acesso do nosso projeto. Vou criar a classe perfil no pacote modelo mesmo. Essa classe tem que ser uma entidade, isso já vimos como fazer no outro treinamento. E precisamos ter um id. Essa classe, além do id, só vai ter um atributo, que é uma string com o nome do perfil. Vou gerar os getters e setters do id e do atributo nome.

Tudo ok. Só que se olharmos no método, ele ainda dá erro, porque a coleção tem que herdar de grantedAuthority, que é uma interface. A classe que representa o perfil também precisamos implementar uma interface do Spring. E a interface que vamos implementar é a grantedAuthority. Nela só tem um único método, que é o getAuthority, para devolvermos qual atributo tem o nome do authority, o nome do perfil.

Ele fica dando o warning porque tem que ter o atributo do serialVersionUID. Vou mandar o Eclipse gerar esse atributo, só para parar de dar erro. Não é obrigatório porque o Java gera o atributo.

Pronto, já ensinei para o Spring que a nossa classe usuário representa, e a classe perfil é a classe que tem perfil de acesso ao usuário. Só faltou um detalhe. Como a classe perfil é uma entidade, preciso dizer qual a cardinalidade do relacionamento, de usuário para perfil. Como um usuário pode ter vários perfis e um perfil pode estar atrelado a mais de um usuário ao mesmo tempo, vai ser um @ManyToMany.

Por padrão, na JPA, todo relacionamento que é ManyToMany, se eu carregar do banco de dados, ele não carrega a lista, porque é lazy, só que eu vou colocar o fetch para ser Eager, porque quando eu carregar o usuário já carrego a lista de perfis, porque vou precisar dos perfis de acesso do usuário.

Já mapeei a classe usuário, a classe perfil, seguindo as interfaces do Spring. Fiz o primeiro passo. Agora, na classe de configuração, depois que configurei as URLs e disse que qualquer outra url tem que estar autenticada, vou dar um enter e continuar, usando and().formLogin(). Existe esse método que é para falar para o Spring gerar um formulário de autenticação. O Spring já tem um formulário de autenticação e um controller que recebe as requisições desse formulário. Então vou chamar esse método porque quero utilizar esse formulário padrão do Spring.

Já está tudo computando. Vou rodar o projeto. Ele vai inicializar, vamos ver se vai dar algum erro. Perfeito. Vou testar isso diretamente no browser só para vermos a página de login. Se eu tentar entrar no localhost:8080/. Se eu der um enter aqui, ele me joga para uma tela de login. O Spring detectou que o / não estava liberado, que eu configurei o endform login, ele redirecionou para a página de login. Nós não criamos essa página no projeto, ela é do próprio Spring. Está até em inglês, e é legal, porque ele colocou o Bootstrap.

Porém, eu não ensinei para ele a lógica de autenticação. Qualquer coisa que eu digitar vai dar um erro. Não tem um provider de autenticação

**Autenticando o usuário**

amos voltar para o nosso código. Na classe securityConfigurations, até então estávamos mexendo nesse configure, que é o configure para parte de autorização das URLs, mas lembre-se que eu tinha comentado que tem outro configure que é da parte de autenticação. É aqui que ensinamos para o Spring como vai ser o processo de autenticação do nosso projeto.

Esse método recebe um parâmetro chamado authenticationManagerBuilder. Se usarmos essa variável, auth., você vai ver que tem vários métodos. Um deles é o UserDetailsService, que é para dizer para o Spring qual a classe, a service que tem a lógica de autenticação. Temos que passar para ele uma classe que está implementando uma interface que é a classe onde estará a lógica de autenticação.

Eu vou passar como parâmetro autenticaçãoService, que vai ser nossa classe com a lógica de autenticação.

Essa classe vai ser uma classe gerenciada pelo Spring, só que não é um controller, não é um repository. Vai ser um service mesmo. Em cima da classe, vou colocar o @Service e importar do Spring.

Para dizermos para o Spring que essa service é a service que tem a lógica de autenticação, também vamos usar uma interface. Vamos implementar a interface UserDetailsService. Lembra que a classe usuário é UserDetails? Já vai dar erro de compilação, porque precisa ter um método, o método loadUserByUsername. Quando entrarmos no formulário de login, digitar o e-mail e clicarmos no botão de login, o Spring vai saber que essa é a classe autenticação e vai chamar esse método. Por isso precisamos da interface. Ele vai procurar esse método, que recebe como parâmetro o e-mail que digitamos na tela de login.

E a senha? Para o Spring, ele só passa o e-mail e nós temos que fazer a consulta no banco de dados filtrando só por isso. A senha ele faz em memória. No geral, em uma aplicação já fazemos a consulta no banco filtrando pelo usuário e pela senha. No Spring security é diferente. Nós buscamos pelo e-mail e o Spring faz a checagem da senha em memória.

Preciso implementar a lógica de fazer a consulta no banco de dados, passando como parâmetro esse e-mail. Como estou numa service, lembra que não é a service que faz a lógica de acesso ao banco de dados. Estamos usando o padrão repository. Então, no nosso pacote repository vamos precisar de um repository para o usuário. Só temos o do tópico e do curso, se não me engano.

Vamos criar uma nova interface no pacote repository, e ela vai ser a usuarioRepository. Lembre-se que todo repository tem que dar um extends JpaRepository<usuário, Long>.

Vamos ter que ter um método que faz a busca pelo e-mail. Ele vai devolver um usuário e o nome do método seguindo aquele padrão de nomenclatura. Ele recebe como parâmetro um string e-mail.

Voltando para minha service, nesse método tenho que chamar o repository. Vou precisar declarar um atributo do tipo usuarioRepository, vou chamar a variável de repository e vou pedir para o Spring injetar o parâmetro @AutoWired. Para fazer a consulta, repository.findbyemail. E passamos como parâmetro username, que foi o Spring que passou de acordo com o que o usuário digitou na tela de login.

Vou guardar o retorno dessa chamada numa variável. Ela devolve um usuário. Vai fazer a consulta e vai devolver o usuário. Na verdade, lembre-se que ele vai fazer a consulta e se o método find não encontrar ele retorna vazio. Vimos isso no TopicosController. Quando fizemos o detalhar, passávamos o id, mas se não existisse o id, ele jogava exception. Começamos a usar o findById, que devolve um optional, e o optional pode ser que tenha resultado ou não. Então, vou precisar fazer o mesmo esquema, porque estou passando o e-mail, mas vai que não existe um usuário com esse e-mail. Eu vou alterar o findbyemail no repository para devolver o optional. Pode ser que ele encontre o usuário, pode ser que não.

Vou trocar o retorno para optional de usuário. Se veio um usuário, é porque está certo o e-mail que você digitou. Simplesmente devolvo o usuário. Se não vier o usuário, preciso dizer para o Spring que o usuário não existe. Lembre-se que dá para fazer um if desse optional. Se estiver presente, devolve. Senão, jogo uma exception para avisar o Spring que o usuário não foi encontrado.

Se chegou ali embaixo, ele digitou um usuário que não existe no banco, throw new usernameNotFoundException. E passo uma string com texto. Essa é a lógica do nosso serviço de autenticação. Quando eu entrar na tela de login e clicar em sign in, o Spring sozinho vai chamar a service.

Agora, voltando para o security Configuration, preciso passar o autenticaçãoService para ele. Só que aqui vou precisar injetar nosso autenticaçãoService. Nessa classe de configuração você consegue também fazer injeções de dependência.

Passei para ele a lógica de autenticação. E compila, porque essa classe nossa implementa a interface do Spring. A princípio, está tudo ok, já ensinei para o Spring como faço a lógica de autenticação. para testar, preciso ter um usuário cadastrado no banco de dados.

Lembra daquele arquivo data.sql? Naquele script de inicialização do banco de dados eu criei um registro para o usuário. O e-mail dele é alura@email,com, a senha é 123456. Porém, em qualquer aplicação deixar a senha em aberto no banco de dados não é uma boa prática. Geralmente o pessoal utiliza algum algoritmo para gerar um HASH dessa senha e salva no banco de dados a senha com o HASH.

Também vamos fazer isso na classe de autenticação. Nós chamamos o auth, o UserDetailsService, só que antes do ponto e vírgula colocamos um passwordEncoder. Conseguimos dizer para ele qual o algoritmo de HASH que vamos utilizar para a senha. Também não adianta nada usar um algoritmo que não é seguro. O Spring suporta alguns algoritmos que são seguros hoje em dia. Um deles é o tal do BCrypt. Posso instanciar um objeto BCryptPasswordEncoder. É uma classe do próprio Spring security que já implementa a geração do algoritmo do HASH, da senha, usando o algoritmo específico.

Só que no nosso data.sql, quando ele vai criar um usuário no banco, ele está jogando a senha em texto aberto. Não posso fazer isso. Tenho que passar um valor que é o HASH no formato do BCrypt. Para facilitar, vou na minha classe SecurityConfigurations e vou dar um system out, vou chamar o método encode e vou passar a senha 123456. Se eu rodar esse main, só estou falando para ele pegar o BCrypt e gerar o HASH da senha.

No arquivo SQL vou trocar a senha pela senha criptografada. Vou voltar no SecurityConfiguration e vou apagar o main, só para termos a tenha gerada. Agora, na teoria, já consigo simular. Vou tentar me autenticar com aquele usuário. Digitando a senha errada, tento logar, ele não permite. Com a senha certa, funciona. Consigo me autenticar no sistema, e a partir de agora consigo chamar os endpoints que estão restritos.

No vídeo anterior, vimos que foi necessário criar uma classe, implementando a interface UserDetailsService do Spring Security. Qual o objetivo dessa classe?

Para indicar ao Spring Security que essa é a classe service que executa a lógica de autenticação

A classe que implementa essa interface geralmente contém uma lógica para validar as credenciais de um cliente que está se autenticando.

Nesta aula, aprendemos que:

* Para utilizar o módulo do Spring Security, devemos adicioná-lo como dependência do projeto no arquivo **pom.xml**;
* Para habilitar e configurar o controle de autenticação e autorização do projeto, devemos criar uma classe e anotá-la com @Configuration e @EnableWebSecurity;
* Para liberar acesso a algum *endpoint* da nossa API, devemos chamar o método http.authorizeRequests().antMatchers().permitAll() dentro do método configure(HttpSecurity http), que está na classe SecurityConfigurations;
* O método anyRequest().authenticated() indica ao Spring Security para bloquear todos os *endpoints* que não foram liberados anteriormente com o método permitAll();
* Para implementar o controle de autenticação na API, devemos implementar a interface UserDetails na classe Usuario e também implementar a interface GrantedAuthority na classe Perfil;
* Para o Spring Security gerar automaticamente um formulário de login, devemos chamar o método and().formLogin(), dentro do método configure(HttpSecurity http), que está na classe SecurityConfigurations;
* A lógica de autenticação, que consulta o usuário no banco de dados, deve implementar a interface UserDetailsService;
* Devemos indicar ao Spring Security qual o algoritmo de hashing de senha que utilizaremos na API, chamando o método passwordEncoder(), dentro do método configure(AuthenticationManagerBuilder auth), que está na classe SecurityConfigurations.

**Por que autenticar via token**

Na última aula, nós aprendemos sobre o Spring security, como proteger nossa API REST liberando endpoints e em outros exigindo autenticação. Porém, a autenticação que fizemos foi a tradicional, com usuário e senha, e o servidor, sempre que o usuário efetua login, cria uma sessão para guardar essas informações. Mas isso não é uma boa prática no modelo REST. O ideal é que a nossa autenticação seja stateless. Se vocês lembram, no curso anterior tivemos uma discussão sobre o modelo REST, e uma das características desse modelo é que toda a comunicação seja de modelo stateless. Ou seja, o cliente dispara uma requisição, leva todas as informações necessárias, o servidor processa, executa o que tem que executar, devolve a resposta e acabou.

Eu tenho um slide mostrando essa ideia da autenticação via session. Em uma aplicação web tradicional, esse é o modelo utilizado. Toda vez que o usuário vai se autenticar no sistema, ele entra no formulário de login, digita o e-mail, a senha, quando faz o login o sistema cria uma sessão e nessa session ele armazena as informações do usuário. Para o servidor conseguir diferenciar um do outro, cada sessão tem id único. Em uma aplicação Java, essa sessão é chamada de jsessionid.

Esse id é devolvido como resposta para o navegador, no formato de um cookie. O navegador armazena isso em um cookie, armazena o id da sessão. Nas próximas requisições que esse usuário disparar nesse navegador, o browser automaticamente envia esse parâmetro como um cookie. Quando chega uma requisição para o servidor, ele verifica se está vindo um cookie Jsessionid. Se estiver, ele recupera o id, com os dados.

Dentro da sessão tem as informações. Isso vai contra um dos princípios do REST, que é de ser stateless. Nesse modelo, para cada usuário que estiver logado na aplicação vou ter um espaço na memória armazenando as informações. Isso consome espaço de memória, e se o servidor cair vou perder todas as sessões. Se eu quiser ter escalabilidade, se eu quiser ter um balanceamento de carga com múltiplos servidores, eu teria problema de compartilhamento.

No modelo REST o ideal é trabalhar com a autenticação de maneira stateless. Com o Spring security é possível fazer isso. Conseguimos explicar para o Spring que não é para ele criar a sessão, que toda vez que o usuário se logar vou fazer a lógica de autenticação, mas não é para criar uma session. Só que aí, nas próximas requisições que os clientes dispararem, o servidor não sabe se está logado ou não, porque não tem nada armazenado. O cliente vai ter que mandar alguma informação dizendo quem é ele, se ele está logado, se tem permissão para acessar. Isso geralmente é feito via tokens. O pessoal costuma usar a especificação JSON web token para fazer esse tipo de autenticação. Ou seja, a cada requisição, o cliente vai mandar um token identificando quem é o usuário que está disparando essa requisição, se ele tem permissão para disparar.

**Configurando autenticação Stateless**

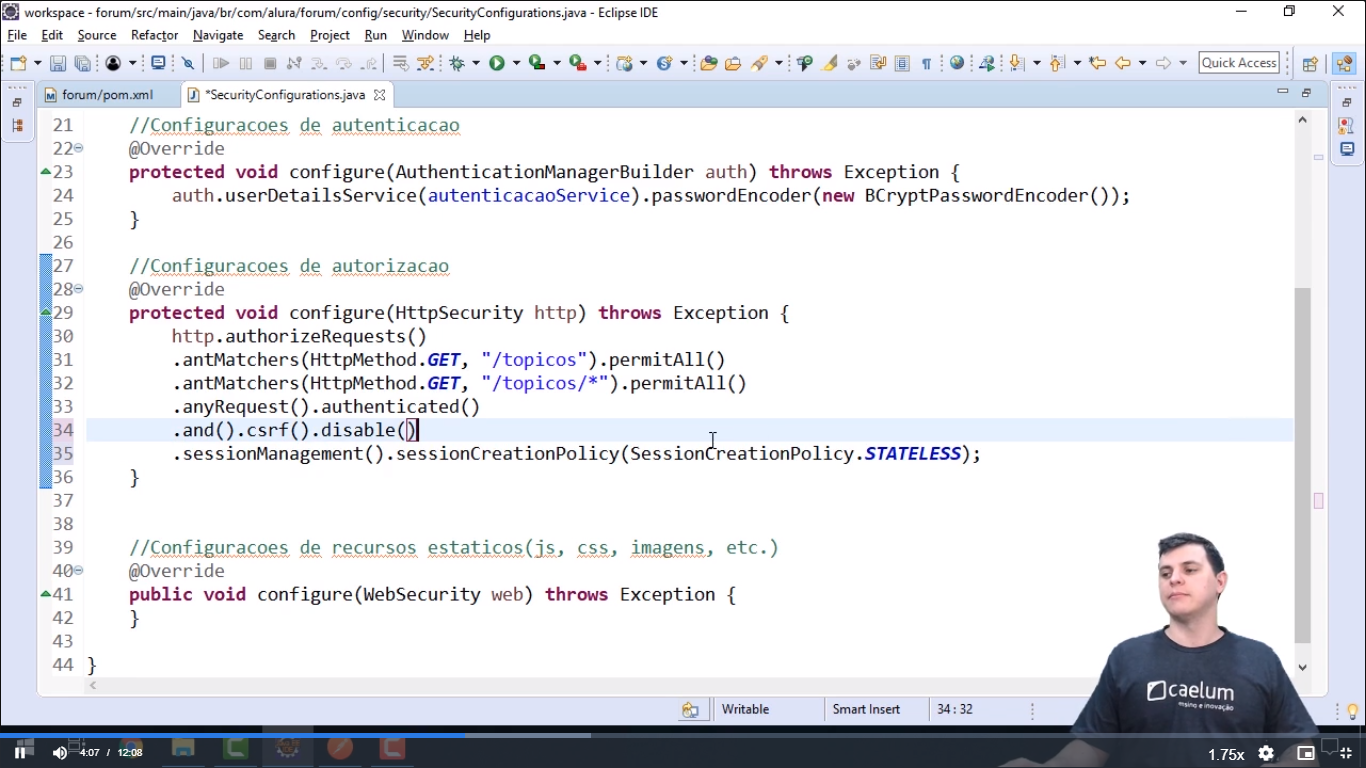
JSON web token, uma tecnologia para geração de tokens e autenticação de maneira stateless.

Para começar, precisamos usar alguma biblioteca em Java que siga esse modelo do JSON web token. Vamos utilizar uma biblioteca chamada jjwt. O primeiro passo é adicionar essa dependência no nosso projeto.

Já com a biblioteca no projeto, precisamos configurar a parte de segurança, naquela classe securityConfigurations, para informar para o Spring security que não vamos mais fazer autenticação usando sessão, mas sim autenticação de maneira stateless. Isso fazemos no método configure, onde estamos configurando as URLs. Embaixo, depois que configuramos as duas URLs e disse que todas as outras precisam estar autenticadas, vamos tirar a parte de and().formLogin(). Não vamos mais usar o formulário de login tradicional, porque senão ele vai criar sessão.

Como estamos fazendo autenticação via token, via JSON, precisamos fazer outra configuração. Precisamos chamar o método csrf.disable. Csrf é uma abreviação para cross-site request forgery, que é um tipo de ataque hacker que acontece em aplicações web. Como vamos fazer autenticação via token, automaticamente nossa API está livre desse tipo de ataque. Nós vamos desabilitar isso para o Spring security não fazer a validação do token do csrf.

Na sequência, sessionManagement. Vamos chamar aqui um método para dizer que não queremos usar sessão. Nós passamos como parâmetro essionCreationPolicy.STATELESS. Com isso, aviso para o Spring security que no nosso projeto, quando eu fizer autenticação, não é para criar sessão, porque vamos usar token.



Mas tem uma desvantagem. Como apagamos aquela linha do formLogin, a partir de agora não temos mais um formulário de login gerado pelo Spring. Isso faz sentido porque as páginas da nossa aplicação não vão ficar na API backend. Vão ficar na aplicação client no frontend. O formulário de login é o cliente que vai fornecer. Ele só vai chamar nossa API passando os dados de login e senha. Só que além de perder o formulário, também perdemos o controller, que fazia a parte de autenticação. Vamos ter que criar um.

Vamos criar uma nova classe e chamar de AutenticacaoController. Esse é o controller onde vai estar a lógica de autenticação. Para dizer que é um controller do Spring, coloco o @RestController e @RequestMapping. Precisamos configurar qual vai ser a url que esse controller vai responder. Vou colocar /auth. Se chegar alguma requisição para /auth, o Spring sabe que é para chamar esse controller, porque é o endereço para autenticação.

Preciso ter um método onde vai ter a lógica de autenticação. Vou criar um método public que devolve o ResponseEntity, vou chamar o método de autenticar. Depois vemos que o vamos receber de parâmetro. E esse método vai ser chamado se a requisição for /auth, e via método @PostMapping. Como é autenticação, estou recendo parâmetros de usuário e senha, preciso que seja via método post.

Esse método vai ser chamado pelo cliente quando ele solicitar os dados de login e senha. A aplicação cliente tem que configurar para chamar esse endereço /auth, a requisição tem que ser via método post, e no corpo da requisição ele tem que mandar o e-mail e a senha. Precisamos receber então o e-mail e a senha. Também quero validar. Vamos fazer validação via bin validação, para garantir que está vindo e-mail e senha. Mas lembre-se que não vou receber os dados soltos e não vou receber uma classe de domínio. Vou receber um form, seguindo o padrão dto.

Seguindo aquele padrão, os dados que chegam do cliente eu recebo numa classe form. Não tem uma classe loginform, preciso criar. Dentro, vou ter só dois atributos, string e-mail e string senha. Quando o cliente fizer a chamada para a nossa API vai ter que mandar dois parâmetros no formato JSON, o e-mail e a senha.

Precisa gerar os setters. E está pronta a lógica para gerar o form. Voltando para o meu controller, o cliente vai chamar /auth via método post, vai mandar um JSON com e-mail e senha, que vou receber dentro do form. Nesse primeiro momento, só para não ficar muita coisa, ainda não vou fazer a geração do token. Só vou fazer um system out para saber se está chegando a senha e o e-mail corretos.

Preciso retornar alguma coisa. Então, ResponseEntity.ok.build. Só para testar se está ok. A ideia é que no meio eu vou precisar pegar o e-mail e a senha, autenticar no sistema, se estiver tudo ok, gero o token. Mas essa parte vamos fazer depois.

No vídeo anterior, tivemos que remover a chamada ao método and().formLogin() da classe SecurityConfigurations. Mas dessa maneira, como um usuário vai se autenticar na API?

O usuário deverá se autenticar por uma página de login fornecida pela própria aplicação cliente

A aplicação cliente deve ter uma página de login, que faz a chamada ao *endpoint* de autenticação da API.

**Gerando tokens com JWT**

Para fazer uma autenticação de maneira programática, manualmente, no Spring security, vamos precisar de uma classe chamada authentication manager. Eu preciso injetar no nosso controller. Vou declarar um atributo que vai ser injetado, do tipo authenticationManager. E vou chamar o atributo de authManager.

[00:55] Porém, tem um pequeno detalhe. Essa classe é do Spring, mas ele não consegue fazer a injeção de dependências dela automaticamente, a não ser que nós configuremos isso. Por algum motivo, ela não vem configurada. Podemos fazer isso na nossa classe SecurityConfiguration. Para a nossa sorte, na classe que estamos herdando já tem um método que sabe criar o authenticationManager. Basta sobrescrevermos esse método.

[01:40] A única coisa que vamos precisar fazer é colocar @Bean em cima do método, porque aí o Spring sabe que esse método devolve o authenticationManager e conseguimos injetar no nosso controller. Vou salvar e voltar para o controller.

[01:57] O próximo passo é pegar no nosso método o authManager e chamar o método authenticate, que vai fazer a autenticação. Para fazer a autenticação, preciso passar para ele os dados de login. Só que não posso passar o loginform ou a senha soltos. O método authenticate recebe um objeto específico. Na linha de cima vamos precisar criar esse objeto. É um objeto do tipo userNamePasswordAuthenticationToken. Vou chamar de dadosLogin. Tenho que dar new nesse objeto passando para ele o login e a senha. Para não deixar isso no controller, vamos fazer aquele esquema que já tínhamos feito no tópico controller. Vamos pegar o próprio form e criar um método converter.

[02:59] Var dar um erro porque não tem o método converter. Vou pedir para o Eclipse criar para mim. Ele recebe como parâmetro o e-mail e a senha. Essa é a classe que o Spring precisa para o authenticationManager. Vou voltar para o controller, agora o authenticate compila, e esse método devolve um objeto do tipo authentication.

[03:33] Quando o Spring chamar essa linha, do authManager.authenticate, o Spring vai olhar as configurações que fizemos e ele sabe que é para chamar a classe authentication services, que chama o usuário repository para consultar os dados do banco de dados. Se der certo, ele vem para a linha de baixo. Se der errado, vai dar um exception. Se der exception, não quero devolver ok ou erro 500. Quero devolver erro400.

[04:08] Eu vou fazer um tratamento, um try catch, vou mover as duas linhas para dentro do try. Tenta fazer a autenticação. Se deu certo, devolve ok. Só que aí não vai ser exception, vai ser authenticationException. Se der erro, retorne badRequest. Com isso já implementei a parte de autenticação. O cliente vai mandar o usuário e a senha, eu chamo o authenticationManager para ele disparar o processo de autenticação. Se estiver tudo ok, ele vai devolver o ok. Senão, vai cair no catch.

[04:56] Só que eu não quero devolver um ok vazio. Quero devolver o token. Antes de devolver o ok, preciso fazer a geração do token, e vou guardá-lo em uma string. Para gerar o token é que vamos usar a biblioteca JWT, que até então só tínhamos colocado no projeto e não utilizado.

[05:17] Mas para não deixar esse código solto, vamos isolar em uma classe service. Vamos criar uma classe chamada TokenService e vamos ter o método gerar token. Na hora de gerar o token vou precisar identificar quem é o usuário, para qual usuário pertence aquele token. Então, nesse método, vou passar o authentication como parâmetro.

[05:47] Dá erro de compilação porque não existe essa variável. Eu vou criar mais um atributo no meu controller do tipo TokenService. Eu quero que o Spring injete para mim. Só que essa classe não existe, vou pedir para ele criar para mim e vou jogar no pacote configSecurity. Nesse pacote vai ter essa classe, que vai ser um service.

[06:21] Voltando para o meu controller, dá erro porque não existe esse método. Vou pedir para ele gerar para mim. Ele vai criar. E aí vou colocar a API do JWT para fazer a geração do token. Ele tem um método que cria um objeto builder onde posso setar informações para ele construir o token.

[06:52] Precisamos acertar alguma coisas. A primeira coisa vai ser o issuer. Quem é que está gerando o token. Vou colocar que foi a API do fórum da Alura, porque aí o cliente consegue identificar quem foi que fez a geração.

[07:17] Outra coisa que preciso setar é o usuário, quem é o dono desse token, quem é o usuário autenticado a quem esse token pertence. E tenho que passar uma string que identifique unicamente meu usuário.

[07:32] Lembra que esse método eu passei como parâmetro authentication? Esse authentication tem um método chamado getPrincipal para conseguirmos recuperar o usuário que está logado. Eu vou jogá-lo em uma variável usuário, que vou chamar de logado. Vai dar um erro de compilação, porque o getPrincipal devolve um object, então tenho que fazer um cast para usuário.

[07:59] No subject, vou colocar logado.getId e vou passar o id. Mas o id precisa que seja string. Também preciso dizer qual foi a data de geração do token. Quando ele foi concedido. E ele trabalha usando a API de datas antigas do Java, então ele está esperando um date. Eu vou criar uma variável ali em cima de date e vou importar do Java.

[08:45] O token também tem uma data de validação, onde ele vai expirar, igual a sessão tradicional, para não ficar infinito, porque isso seria um risco de segurança. E tenho que passar uma data. Eu poderia pegar a data hoje, somar com trinta minutos, mas esse tempo, para não ficar no código, vou injetar em uma propriedade do application.properties chamada fórum.jwt.expiration= e passei um tempo em milissegundos. Coloquei um dia, só para ficar mais fácil no teste. Na prática, o ideal é que o tempo seja menor.

[10:12] Tem uma senha também que depois vou mostrar onde vamos utilizar. Voltando ao TokenService. Tendo essa propriedade, como injetar? Vou declarar um atributo. Para injetar não vai ser um Autowired, porque ele procura um bean que está configurado, e no caso é uma string. Para injetar coisas, parâmetros do Application.properties, usamos a anotação @Value, e ela recebe como parâmetro o nome da propriedade.

[10:56] O Spring vai no application.properties, vai ver quem é a propriedade, pegar o valor dela e injetar aqui. Já vou aproveitar e injetar aquela senha. Vou jogar em um atributo chamado secret.

[11:15] A data de expiração não é o expiration, porque o expiration é o tempo em milissegundos. Tenho que pegar aquele tempo e somar com a data atual. Então, vou criar outro date. Posso passar como parâmetro hoje.getTime, para pegar os milissegundos da variável hoje, e somar com a variável expiration, porque aí ele vai fazer a soma dos dois milissegundos e criar a nova data. É justamente esse data expiração que vou passar como parâmetro.

[11:57] A última coisa que precisamos setar é: tem um método chamado signwith. Pela especificação JSON webtoken, o token tem que ser criptografado. Preciso dizer para ele quem é o algoritmo de criptografia e a senha da minha aplicação, que é usada para fazer a assinatura e gerar o REST da criptografia do token. É aqui que vou injetar o secret, que está no application.properties. A ideia é pegar um programa que gera uma string aleatória gigantesca e usar como senha. Para não ficar isso no código, podemos colocar no application.properties.

[12:45] Só preciso dizer qual é o algoritmo, e aí existe a classe signatureAlgorithm.HS256. No final, um compact para ele compactar e transformar isso em uma string. Ele dá um erro porque está fazendo operações com o date, da API de datas do Java antigo, mas esse é o jeito mais simples, não vai ter nenhum problema.

[13:27] Terminei. Consegui fazer a lógica para gerar o token. Para testar, vamos no controller. Já chamei o service e mandei ele criar um token. Vou dar um system out só para ver se está gerando o token certo. Para testar, vou usar o Postman. Vou disparar a requisição POST. Deu algum erro no nosso projeto na hora de pegar a data.

[14:27] O expiration estava como string, tem que ser um long. Agora voltou 200, deu o system out. No próximo exercício, vamos ver como no controller podemos fazer para devolver esse token, porque aqui eu só fiz o system out para ver se está tudo ok.

No último vídeo, vimos que é possível fazer injeção de dependências de propriedades do arquivo **application.properties**. Dos exemplos de código abaixo, qual deles faz a injeção de propriedades de maneira correta?

@Value(“${forum.jwt.secret}”)

private String secret;

Alternativa correta! A anotação @Value deve declarar o nome da propriedade como String, utilizando a *expression language* ${}.

**Retornando o token para o cliente**

 Continuando, nós já fizemos a geração do token. Agora, ao invés de só devolver um 200, preciso levar esse token junto como resposta. Mas lembram que estávamos usando um padrão de criar um dto e dentro do dto encapsular as informações? Aqui vamos seguir o mesmo padrão.

[00:29] Vou apagar a linha do system out e no ResponseEntity.ok, no corpo da resposta, quero que ele devolva um objeto, que no caso vai ser um new Tokendto. Vamos passar como parâmetro o token que foi gerado.

[00:50] Além do token, também precisamos dizer para o cliente qual o tipo de autenticação, como ele vai fazer a autenticação nas próximas requisições. No http existe uma parte que fala sobre autenticação. Tem vários métodos. Um deles é o bearer. Junto com o token, eu vou mandar para ele o tipo de autenticação que ele tem que fazer nas próximas requisições. Depois vamos ver como entra o bearer e como funciona.

[01:35] Ele está reclamando, porque não tenho a classe tokendto. Vamos criar no nosso pacote dto. Como todo dto, preciso ter os getters.

[02:53] Vamos testar então se está funcionando certinho. Vou disparar a mesma requisição post /auth, com os dados de login. Se eu clicar na aba body, foi devolvido o token e o tipo, que é bearer. Esse token vai voltar para o cliente, e o cliente é que é responsável por guardar em algum lugar, porque nas próximas requisições que ele disparar, ele vai ter que levar esse token a um cabeçalho autorization, que é o cabeçalho responsável por cuidar da parte de autorização. Nesse cabeçalho, preciso dizer qual o tipo de autenticação. No próximo vídeo vamos ver como mandar esse token e fazer a autenticação.

No último vídeo, vimos que ao devolver o token para o cliente, foi enviado juntamente outra informação, chamada **Bearer**. A que se refere essa informação?

É o tipo de autenticação a ser feita pelo cliente com o *token* que lhe foi devolvido

**Bearer** é um dos mecanismos de autenticação utilizados no protocolo HTTP, tal como o Basic e o Digest.

Nesta aula, aprendemos que:

* Em uma API Rest, não é uma boa prática utilizar autenticação com o uso de *session*;
* Uma das maneiras de fazer autenticação *stateless* é utilizando *tokens* **JWT** (*Json Web Token*);
* Para utilizar JWT na API, devemos adicionar a dependência da biblioteca **jjwt** no arquivo **pom.xml** do projeto;
* Para configurar a autenticação *stateless* no Spring Security, devemos utilizar o método sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
* Para disparar manualmente o processo de autenticação no Spring Security, devemos utilizar a classe AuthenticationManager;
* Para poder injetar o AuthenticationManager no *controller*, devemos criar um método anotado com @Bean, na classe SecurityConfigurations, que retorna uma chamada ao método super.authenticationManager();
* Para criar o *token* JWT, devemos utilizar a classe Jwts;
* O *token* tem um período de expiração, que pode ser definida no arquivo **application.properties**;
* Para injetar uma propriedade do arquivo **application.properties**, devemos utilizar a anotação @Value.

**Recuperando o token do header Authorization**

Na última aula vimos como fazer a geração do token, usando a biblioteca do JSON web token, e criamos nosso controller para fazer a parte de autenticação, que recebe a requisição do cliente com o e-mail e a senha, valida no banco de dados se os dados estão corretos, e se estiver gera o token devolvendo dentro do dto.

[00:34] A primeira coisa que o cliente vai fazer é disparar uma requisição do tipo POST para aquele endereço /auth, que é o endereço de autenticação. E no corpo da requisição ele tem que levar o JSON com e-mail e senha. Eu vou disparar a requisição e no retorno vai voltar como resposta o token e o tipo de autenticação que o cliente vai ter que fazer.

[01:33] A partir dessa requisição, o cliente da API é responsável por pegar o token que voltou na resposta e armazenar em algum lugar. Se for uma aplicação frontend, geralmente ele vai usar um cookie ou um session storage, ou outro lugar em memória. E nas próximas requisições que ele disparar para a nossa API para recursos protegidos ele vai ter que levar esse token. Como ele faz isso?

[02:01] Vou mostrar um exemplo. Imagine que ele acabou de se autenticar e pegou o token. Agora quero chamar aquele endpoint DELETE para /tópicos/2. Quero excluir o tópico de id 2. Configuramos isso para ser restrito. Se eu disparar a requisição sem passar nada, vou receber um 403. Para ele disparar a requisição, ele tem que adicionar mais um cabeçalho, que é o authorization. A ideia é colocar o cabeçalho, com o valor bearer, e o token. Nas próximas requisições, o cliente tem que sempre mandar o cabeçalho authorization com o token. Se eu disparar a requisição agora, também vai dar 403, porque ainda não implementamos a lógica. Esse vai ser o objetivo dessa aula.

[03:17] Vamos voltar para o nosso projeto. A ideia é que essa lógica tem que rodar antes de cair no meu controller, no meu código. Tenho que interceptar a requisição e executar a lógica. Para fazer isso, vamos criar um filtro. Vou clicar no nosso pacote security, vou criar uma classe, chamar de autenticaçãoViaTokenFilter. No caso, ela vai ser um filter.

[03:55] No Spring, podemos herdar essa classe de outra chamada OncePerRequestFilter, que é um filtro do Spring chamado uma única vez a cada requisição. Dá um erro, porque tem um método abstrato que precisamos implementar. Vou adicionar esse método, chamado doFilterInternal.

[04:15] O que temos que fazer aqui nesse método? Temos que colocar nossa lógica para pegar o token do cabeçalho, verificar se está ok, autenticar no Spring, e a última linha que tem que ter nesse método precisa pegar o terceiro parâmetro, que é o filterChain.doFilter, passando request, response.

[04:40] Essa é a linha para falar para o Spring que já rodamos o que tínhamos que rodar nesse filtro. Só que antes vou precisar recuperar o token no cabeçalho, validá-lo, e se estiver ok autentificar o usuário para o Spring. Eu tenho que autenticar porque como a nossa autenticação é stateless, não existe mais a ideia de usuário logado. Então, quando o cliente disparou a requisição para se autenticar, passando e-mail e senha, eu gerei um token, devolvi para ele, mas eu não guardei esse token no lado do servidor. Minha API não sabe se aquele usuário está logado. Ela tem os tokens que foram gerados, mas ela não sabe quem está ou não está logado. Na próxima requisição que esse cliente disparar, nossa API não tem como saber se esse cliente está autenticado ou não, porque não existe esse conceito de autenticação.

[05:43] A autenticação é feita para cada requisição. Toda requisição que chegar para a nossa API, nós pegamos o token, autenticamos o usuário, se estiver ok o token, executamos a requisição, devolvemos a resposta, acabou o assunto. Na próxima requisição a API nem lembra mais quem é esse cliente. Tenho que pegar o token do cabeçalho, autenticar de novo, rodar o request e devolver a resposta. Por isso é por requisição.

[06:19] Eu preciso pegar o token. O primeiro passo da nossa lógica é recuperar o token do cabeçalho. Para fazer isso, vou criar uma variável string token = recuperarToken. Nesse método passo o request, e ele já me devolve o token. O request tem um método chamado getHeader. Nele, passo o nome do cabeçalho que quero recuperar. No caso, vimos que é authorization.

[07:17] Vou guardar isso em uma variável. Só que aí pode ser que esse cabeçalho não esteja vindo, que ele não esteja nesse formato. Preciso verificar se está correto ou não. Vou fazer um if, se o token for igual a nulo, ou se token está vazio, ou se o token não começa com bearer, eu vou devolver nulo. É só para verificar se está vindo o token, se não é vazio, e se começa com bearer. Se nenhuma dessas condições forem ok, eu devolvo nulo. Senão, devolvo o token. Só que se eu devolver o token assim vai o conteúdo inteiro. Mas eu não quero esse começo. Quero só o token em si, então vai ser token.substring(7, tolen.length). Sete porque vai começar a pegar a partir do espaço até o final da string, que é o conteúdo do token.

[09:00] Devolvi isso. O próximo passo é validar o token, ver se ele está correto. Aí vamos ter que usar nossa biblioteca, o jjwt, para fazer essa validação. Se estiver ok, o terceiro passo é falar para o Spring autenticar o usuário. Mas como são vários passos e é muito código, vamos quebrar isso e fazer aos poucos. Neste vídeo, só vamos pegar o token e imprimir no console para ver se está chegando corretamente.

[09:40] Vamos testar no Postman, mandando o cabeçalho bearer e o token que foi devolvido anteriormente. Não funcionou porque tem mais um detalhe. Eu só criei o filter, mas não tem anotação nenhuma. Embora eu tenha herdado de uma classe, ele não registrou. Precisamos registrar esse filtro para o Spring.

[10:10] Para fazer isso, não é via anotação. Tem que ser na nossa classe Security Configuration. No nosso método configure, que tem as URLs, depois que eu configurei que a autenticação é stateless, vou colocar mais uma sentença, o addFilter. Só que não posso chamar isso, porque o Spring internamente já tem o filtro de autenticação. Ele precisa saber qual a ordem dos filtros, quem vem antes. Por isso, tem que ser o método addFilterBefore. Passo para ele quem é o filtro que quero adicionar e antes de quem esse filtro virá. Depois, damos um new AutenticacaoViaTokenFilter(), UsernamePasswordAuthenticationFilter.class. Esse é o token que já tem no Spring por padrão. Vou falar para o nosso filtro rodar antes dele.

[11:22] Vou salvar. E agora volto no Postman, disparo a requisição. Veio exatamente o token certinho. Vou fazer outro teste mandando o cabeçalho vazio. Disparo a requisição. Ele imprime nulo. Está funcionando certinho.

No último vídeo, vimos que precisamos criar um filtro, que vai conter a lógica de recuperar o *token* do cabeçalho Authorization, validá-lo e autenticar o cliente. Vimos também que esse filtro precisa ser registrado no Spring.

Qual anotação foi utilizada para registrar o filtro?

Não existe anotação para registrar o filtro

O filtro deve ser registrado via método addFilterBefore, na classe SecurityConfigurations.

**Validando o token**

Nós já conseguimos recuperar o token do cabeça authorization e imprimimos para ver se estava chegando certinho. O próximo passo é ir ao nosso filter. Preciso verificar se o token está válido, se é um token gerado pela aplicação ou se algum cliente mandou um token, mas digitou qualquer coisa aleatoriamente.

[00:33] Para fazer isso, preciso usar aquela API naquela biblioteca que estamos usando, do JSON web token. Tínhamos isolado o código dela naquela classe token service. Nessa a classe vamos criar um método que recebe um token e me devolve um boolean me dizendo se está válido ou não.

[00:52] Eu vou simplesmente fazer a chamada para essa classe e guardar o boolean que foi devolvido. Vou criar uma variável boolean, chamar de valido = tokenservice.istokenvalido(token).

[01:21] Dá um erro porque não existe esse atributo. Vou precisar criar um atributo do tipo TokenService. A ideia seria injetar esse TokenService. Porém tem um problema. Nesse tipo de classe não conseguimos fazer injeção de dependências. Não dá para colocar um @AutoWired, até porque na classe de security Configuration nós que instanciamos manualmente a classe.

[01:58] Eu posso receber via construtor. Já que não posso injetar pelo atributo, vou injetar da mesma maneira, só que via construtor. Quem for dar new no nosso filter vai ter que passar o token service como parâmetro. Vou salvar. Só que na verdade quem está dando new na classe token filter somos nós mesmos pela classe security Configuration.

[02:32] Mas aqui estou na classe security Configuration. Nessa a classe eu consigo fazer injeção de dependências. Resolvemos o problema. Não consigo injetar no filter, mas na classe Configuration eu consigo. Na hora de dar new no filter, eu passo o token service que foi injetado. Problema resolvido. Ele está só reclamando porque o método isTokenValido não existe. Temos que criar.

[03:10] Aqui temos aquele método para gerar o token. Preciso ter esse método para fazer a validação, para validar se o token que está chegando está ok ou não. Para fazer isso, vamos usar de novo o tal de jwts. Só que não vou chamar o builder, porque não quero criar um novo token. Vou chamar o método parser, que é o método que tem a lógica para fazer o passe de um token. Você passa para ele um token, ele vai descriptografar e verificar se está ok.

[03:43] Na sequência, temos que chamar primeiro setSigningKey. Tenho que passar aquele secret da nossa aplicação, que é a chave que ele usa para criptografar e descriptografar. Tem um método chamado parseClaimsJws. Esse é o método que vamos chamar passando como parâmetro o token.

[04:18] Esse método devolve o Jws claims, que é um objeto onde consigo recuperar o token e as informações que setei dentro do token. Mas quando eu fizer essa chamada, se o token estiver válido, ele devolve o objeto. Se estiver inválido ou nulo, ele joga uma exception. Eu vou fazer um try catch, vou colocar o código dentro do try. Se ele rodou tudo ok, o token está válido. Se chegou na linha de baixo é porque o token está válido, retorna true, porque não quero recuperar nenhuma informação do token nesse método. Se deu alguma exception, ele vai entrar no false.

[05:27] Por enquanto, a única coisa que preciso fazer é devolver se está válido ou se não está válido. Só isso. Salvei. Vou voltar para o meu filter. Nesse exercício, só vamos validar se está ok ou não. Vamos fazer aquele teste via Postman. Vou mandar um token correto, ver se está válido e se vai imprimir true. Depois mando um token aleatório, ver se vai imprimir falso. E vou mandar vazio.

[06:15] Preciso do cabeçalho authorization, bearer, crio a requisição para fazer o DELETE. No Postman, inclusive, tem um cabeçalho authorization. Posso ir direto por ali. Ele já seta automaticamente o cabeçalho, não preciso digitar. Vou disparar a requisição. Deu proibido, porque ainda não fiz a lógica de autenticar no Spring, mas para mim o que interessa é que imprimiu true. Vou disparar de novo com o token errado. Imprimiu false. Vou disparar vazio. Imprimiu false.

Por que não é possível fazer injeção de dependências com a anotação @Autowired na classe AutenticacaoViaTokenFilter?

Porque ela não é um *bean* gerenciado pelo Spring

O filtro foi instanciado manualmente por nós, na classe SecurityConfigurations e portanto o Spring não consegue realizar injeção de dependências via @Autowired.

**Autenticando o cliente via Spring Security**

Agora que já conseguimos recuperar o token do cabeçalho e fazer a validação, podemos finalmente ir para o último passo. Já que o token está ok, vamos pedir para o Spring considerar que o cliente que está disparando essa requisição está autenticado. E caso o token esteja invalido não vamos chamar esse código para autenticar.

[00:42] Agora, eu preciso fazer um if. Se o token estiver válido, tenho que autenticar o usuário. Se estiver inválido, não vai entrar no if, ele vai chamar o filterChain e vai seguir o fluxo da requisição, barrando o usuário. Só para não ficar com essa lógica, vou criar um método privado também autenticarCliente(token). Vou criar esse método. Agora conseguimos implementar a lógica.

[01:35] Diferente do AutenticacaoController, que utilizamos o authentication manager, no filtro não vou usar isso, porque ele é para disparar o processo de autenticação, com usuário e senha. Aqui não quero fazer autenticação via usuário e senha, porque esse processo já foi feito. Só quero falar para o Spring autenticar o usuário. Para fazer isso, o Spring tem uma classe chamada SecurityContextHolder.getContext().setAuthentication. Esse é o método para falar para o Spring considerar que está autenticado. Só que aí preciso passar os dados do usuário. Então, na hora de passar esse método preciso passar o objeto authentication, que tem as informações do usuário que está logado.

[02:51] Ele recebe um objeto authentication, e preciso criar uma variável do tipo authentication, é aquela mesma variável que usamos no AutenticacaoController. Podemos dar um new, mas aí vamos usar outro construtor. Em um passo os dados do usuário, as credenciais, que vai ser nulo, porque já validei antes, e preciso passar também os perfis de acesso dele.

[03:55] Só vai ter um probleminha, porque não temos a variável usuário. Para dar new no authentication preciso do usuário. De onde ele veio? Precisamos recuperar. A única informação que tenho na mão é o token. E não sei se vocês lembram, mas na classe TokenService, quando escrevemos aquele código para gerar o token, nós setamos uma linha setSubject(logado.getId().toString()). Dentro do token, temos o id do usuário. Se eu tenho o id, consigo carregar o objeto do banco de dados.

[04:41] Eu vou fazer o seguinte. Estou com o token, dentro do token vou ter o id do usuário, preciso recuperar esse id, com algo como Long idUsuario = TokenService.getIdUsuario(token). Vou criar mais esse método no TokenService, em que passo um token e ele me devolve um id que está dentro desse token.

[05:21] Vou pedir para o Eclipse gerar o método para mim. E aí, como faço para recuperar os dados do token? É parecido com o método isTokenValido. Vamos precisar desse código, da classe que faz o parser do token. Vou colocar jwts.parser().setSigningKey(this.secret).parseClaimsJws(token). Esse objeto tem um método na sequência chamado getBody, que devolve o objeto do token em si. Vou chamar esse método e vou pedir para ele guardar isso numa variável local.

[06:08] Ele devolve esse objeto do tipo claims, que é uma classe também da API do JSON web token. Na próxima linha, dentro desse objeto claims, tenho vários métodos get para recuperar as informações do token. Na hora em que geramos o token, setamos o subject. Então aqui consigo chamar o getSubject, para pegar o id de volta.

[06:42] Só que dentro do token tudo é string, mas eu não quero isso como string, quero como long. Mas eu sei que é um long que está lá dentro, que é o id do usuário, então vou fazer um parse aqui. Com isso eu consigo recuperar o id do usuário que está setado dentro do token.

[07:04] Vou voltar para o meu filter. Agora já tenho o id do usuário. Só que eu preciso do objeto usuário completo. Na sequência, preciso carregar esse usuário do banco de dados. Para carregar do banco de dados tem o repository. Mas não tenho acesso a ele. Vou ter que declarar mais um atributo. Vou chamar a variável de repository. Tem aquele problema de não conseguir injetar. Vou ter que passar esse parâmetro para o construtor, na hora de dar new no nosso filtro. Lembre-se que quem está dando new nesse objeto é a classe security configurations.

[08:00] Eu não tenho essa variável, mas consigo injetar. Vou declarar mais um atributo na classe de configuração, que é o usuarioRepository. Só não posso esquecer o @AutoWired, senão o Spring não vai injetar.

[08:17] Injetei o usuarioRepository, dei o new. No meu filter, quando dei new, eu injetei no construtor o token service e o usuarioRepository. Agora, consigo recuperar o objeto usuário da seguinte maneira: usuário = repository.getOne(idUsuario). Pronto, recuperei o usuário. É ela que passo como parâmetro. Vou salvar, e agora está tudo compilando.

[09:33] Peguei o id do token, recuperei o objeto usuário passando o id, criei o usernameauthenticationtoken passando o usuário, passando nulo na senha, porque não preciso dela, passando os perfis, e aí por fim chamei a classe do Spring que força a autenticação. Essa autenticação é só para esse request. Na próxima requisição ele vai passar no filter de novo, pegar o token e fazer todo o processo. A autenticação é stateless. Em cada requisição eu reautentico o usuário só para executar aquela requisição.

[10:10] No nosso método principal do filter só chamo o autenticar se o token estiver válido. Se não estiver, não vai autenticar, vai seguir o fluxo da requisição e o Spring vai barrar. Agora está tudo implementado. Vamos fazer nosso teste.

[10:23] Vou até o Postman e vou testar tudo de novo, do zero. Vou fazer o localhost:8080/auth, método POST. Quero autenticar. Vou primeiro fazer o login. No body vou colocar raw. Vou passar o JSON com e-mail e senha. Vou disparar essa requisição, que é a primeira que o cliente vai fazer, para se autenticar. Disparou, devolveu 200 e voltou o token. Vou copiar o token, criar uma nova requisição do tipo DELETE para o tópico 1. Vou ver o que vai acontecer. Ele me dá o 403.

[11:31] Agora vou na aba authorization e vou colocar que a autorização é via bearer token, vou passar o token, mas vou trocar um caractere, vou passar um token inválido. Ele me dá um 403. Vou passar o token válido e ele me deu inválido, porque não posso usar o getOne, tem que ser o findById. O getOne não carrega o objeto na JPA, só traz o proxy, na hora de carregar os perfis deu erro.

[12:51] Testando de novo, ele me dá o código 200. Consegui deletar o tópico. Finalizamos nossa autenticação utilizando token, com aquela biblioteca jjwt do Java integrada ao Spring security. É um pouco chata essa parte, porque tivemos que criar o filter, e tem todo o processo de pegar o token do cabeçalho, validar, forçar autenticação via Spring security, registrar o filtro no security Configuration, injetar os parâmetros para passar para o filter, porque no filtro não posso receber via injeção de dependências, teve que criar o AutenticacaoController, o token service. É muito código, é chato, mas é aquele código que você faz uma vez só. E é um código de infraestrutura, não estou implementando uma regra do projeto. Você não precisa decorar, saber de cabeça. Você pega um de exemplo e joga no projeto, fazendo adaptações.

[14:24] Com isso, conseguimos fazer nossa autenticação de maneira stateless, sem usar sessão, seguindo a ideia do modelo REST. A API REST não guarda estado de autenticação, então a cada request eu mando o token, reautentico. Fica mais leve, mais fácil de ter escalabilidade na nossa API. Por hoje, esse era o assunto. Na próxima aula vamos ver sobre monitoramento, uma coisa importante que preciso ter em uma API REST, ainda mais se eu estiver trabalhando com aquele modelo de arquitetura de micro serviços.

Vimos no último vídeo que foi necessário indicar ao Spring que o cliente está autenticado. Por que essa autenticação foi feita com a classe SecurityContextHolder e não com a AuthenticationManager?

Porque a classe AuthenticationManager dispara o processo de autenticação via *username*/*password*

A classe AuthenticationManager deve ser utilizada apenas na lógica de autenticação via *username/password*, para a geração do *token*.

Nesta aula, aprendemos que:

* Para enviar o *token* JWT na requisição, é necessário adicionar o cabeçalho Authorization, passando como valor Bearer token;
* Para criar um filtro no Spring, devemos criar uma classe que herda da classe OncePerRequestFilter;
* Para recuperar o *token* JWT da requisição no *filter*, devemos chamar o método request.getHeader("Authorization");
* Para habilitar o filtro no Spring Security, devemos chamar o método and().addFilterBefore(new AutenticacaoViaTokenFilter(), UsernamePasswordAuthenticationFilter.class);
* Para indicar ao Spring Security que o cliente está autenticado, devemos utilizar a classe SecurityContextHolder, chamando o método SecurityContextHolder.getContext().setAuthentication(authentication).