**Curso Angular com REST e SpringBoot**

Empresa: AlgaWorks

Instrutores: Thiago Farias e Normandes.

Pre-requisitos: Java básico, html básico, css, typeScript.

**1. Introdução**

A primeira parte do curso ensina a desenvolver o back-end do sistema, utilizando SpringBoot e implementando REST. Na segunda parte desenvolveremos o front-end utilizando Angular.

**SOFEA** - Arquitetura de Front-end orientada a Serviço

Toda a lógica de apresentação do sistema fica no cliente. A renderização das telas ocorre na máquina do usuário. O servidor fica responsável somente pelo processamento do back-end.

REST (Transferência de estados representacional) - É uma forma de integrar sistemas ou camadas de um mesmo sistema, como no nosso caso, o back-end e o front-end, que podem ser consideradas duas aplicações diferentes.

O REST é uma arquitetura, que segue algumas regras e princípios, para que seja implementada com sucesso.

API REST – É o ponto de acesso entre o front-end e o back-end que implementa REST.

**2. O projeto desenvolvido no curso**

A tela inicial do sistema é a tela de login, que valida o login e senha de acesso. A tela seguinte lista lançamentos de um sistema financeiro, com recurso de paginação, em que se pode lançar débitos e créditos. A tela possui um menu intercambiável lateral para acesso aos dados do usuário, cadastrar usuário ou para efetuar log out. Disponibiliza recursos de pesquisa de lançamentos e um CRUD completo, além de controle de permissões de acesso a recursos.

A paginação só busca os dados que estão na página. A cada mudança de página o sistema busca novos dados. Dessa forma o sistema fica mais leve.

Os dados trafegados entre o servidor e o cliente serão no formato JSON.

As telas do sistema são responsíveis, se adaptam para os formatos mais utilizados.

**3. Ambiente de desenvolvimento**

O SpringBoot controla todas as ferramentas do Spring que utilizaremos.

O STS (Spring Tool Suite) é praticamente o Eclipse adaptado para o uso do Spring. Pode-se utilizar qualquer outra IDE que contenha plugin para Spring. O STS oferece algumas funções que os outros editores pode não conter, como por exemplo autocomplete para funções específicas do Spring.

No site <https://start.spring.io>, pode-se gerar a estrutura inicial do seu projeto.

A ferramenta de teste para o back-end é a POSTMAN ([www.getpostman.com](http://www.getpostman.com)). Ela faz requisições http para buscar dados no banco, via beck-end da aplicação, de forma direta, sem que as telas do sistema estejam prontas.

O banco de dados utilizado é o MySQL mas pode-se utilizar outros.

O site [www.mocky.io](http://www.mocky.io) pode ser utilizado para testar APIs REST. (veja **Figura 1**)

4. **O POSTMAN**

Antes de começar a fazer os testes, recomenda-se criar a Collection (pasta) que será utilizada para armazenar os arquivos de teste.

Utilize o mocky.io para gerar a URL a ser testada no POSTMAN.

A **figura 2** mostra o resultado do teste efetuado a partir da URL gerada no mocky.

Quando quiser gravar o teste no POSTMAN, lembre-se de selecionar a Collection que deseja guardar seus testes.

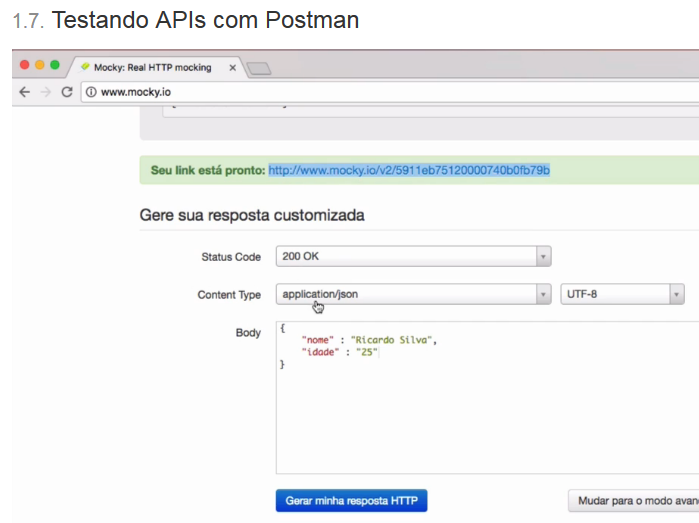


Figura 1. Utilizando o mocky para gerar resposta html para teste no POSTMAN.



Figura 2. Resultado sendo mostrado no POSTMAN

O protocolo de comunicação entre o front-end e o back-end é o HTTP. Ele envia requisições (request) e obtem respostas do servidor (response), identificadas por meio de códigos predeterminados. Exemplo: se o servidor retorna código 200, significa que não houve erro (OK), 201 quando consegue criar um novo elemento no servidor (ex.: cadastro). Se 404 – not found, se 500 – erro no servidor...

O que é um RECURSO?

É todo elemento que tem informações e uma maneira de identifica-lo. Ex.: uma url que passa como parâmetro um ID, retorna informações (recurso) do banco, referente ao identificador informado.

**5. Representações de um recurso**

Os recursos retornados do servidor, podem vir em vários formatos (Ex.: JSON, XML, IMAGE...). No caso do nosso projeto, trabalharemos com formato JSON para retorno. Para informar ao protoloco http que utilizaremos o formato JSON, é necessário passar essa informação pelo Header do protoloco http.

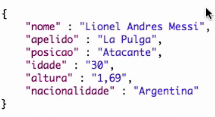


Figura 3 – formato JSON.

**6. Criando o projeto**

No STS - file -> New Spring Starter Project.

Type: Maven

Group: com.algaworks.algamoney-api

Artifact: algamoney-api

Package:com.example. algamoney.api

Precione o botão NEXT e selecione as dependências necessárias para o projeto.

- web (já vem com o Tomcat embarcado)

- jpa

- DevTools (possibilita deploy automático no momento que efetuar alteração no arquivo)

O arquivo pom.xml já vem com todas as dependências do Spring.

Precione FINISH

Na estrutura do projeto já estará criada a classe com o método main, que serve para inicializar o projeto.

**7. Conexão com o banco de dados**

Na pasta src/main/resources, abra o arquivo application.properties. É neste arquivo que configuramos a conexão com o BD, com as seguintes linhas de comando, para conexão no Postgresql:

spring.jpa.database=POSTGRESQL

spring.datasource.url=jdbc:postgresql://curio.sct.embrapa.br:5432/algamoneyapi

spring.datasource.username=pgsql

spring.datasource.password=pgsql

Acrescente esta dependência no arquivo pom.xml:

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

Execute a aplicação para verificar se não ocorrerá erro de conexão.

**8. Criando as tabelas do banco a partir do código do sistema** (aula 3.3)

Utilize este recurso somente para criar o protótipo. Não é aconselhável manter no ambiente de produção.

Para utilizar este recurso é necessário incluir mais uma dependência no pom.xml

<!-- Migracoes - Flyway -->

<dependency>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-core</artifactId>

</dependency>

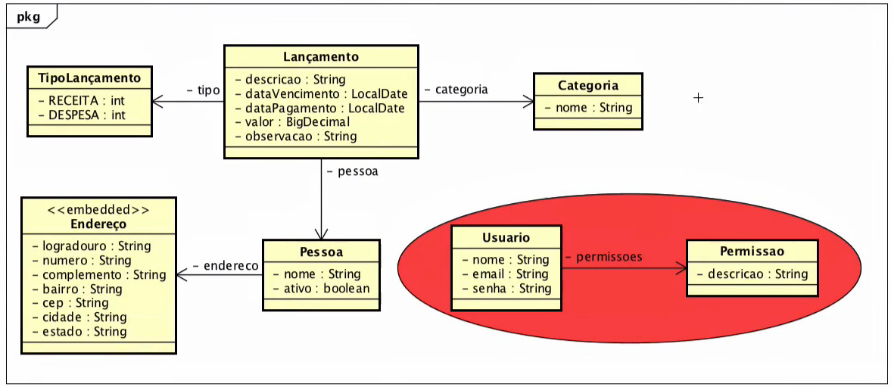


Figura 3 – diagrama do sistema.

Iniciaremos criando a tabela categoria. Na pasta "src/main/resources", crie a pasta bd e dentro dela a pasta migration (Figura 4).



Figura 4 – pasta para os scripts que criam as tabelas no bd.

Dentro da pasta "migration", crie um arquivo com o nome " **V01\_\_criar\_e\_registrar\_categorias.sql**". Deve iniciar-se com letra maiúscula seguida do número da versão. Após o número da versão deve-se colocar obrigatoriamente dois underscore (\_\_). O nome do arquivo não deve conter espaços em branco, coloque undercore para separar as palavras e a extensão deve ser "sql". Dessa forma o Flyway consegue identificar.

Neste arquivo coloque comandos SQL. No caso para criação da tabela categoria, no postgresql, será:

CREATE TABLE public.categoria (

codigo serial NOT NULL,

nome VARCHAR(50) NOT NULL,

CONSTRAINT codigo\_pkey PRIMARY KEY (codigo)

);

Depois de criada a tabela, se futuramente precisar fazer uma alteração na estrutura da tabela, deve-se criar um novo arquivo, alterando a versão (Ex.: de V01\_\_ para V02\_\_), acrescentando a alteração.

Pode-se incluir também o script dos primeiros registros da tabela:

INSERT INTO categoria (nome) values ('Lazer');

INSERT INTO categoria (nome) values ('Alimentação');

INSERT INTO categoria (nome) values ('Supermercado');

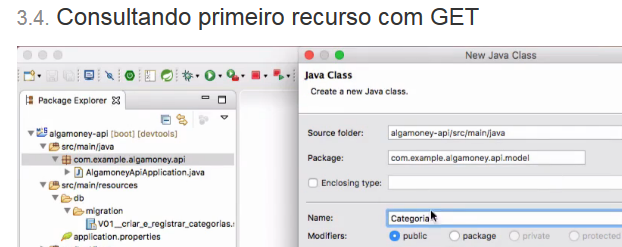
INSERT INTO categoria (nome) values ('Farmácia');

INSERT INTO categoria (nome) values ('Outros');

Execute um comando Select no BD para verificar se realmente a tabela e os registros foram criados.

**9. Criando as primeiras classes**

Na pasta com.example.algamoney.api crie o pacote **model** e dentro dele a classe Categoria.



**Figura 4** – criando a classe Categoria.

@Entity

@Table(name="categoria")

**public** **class** Categoria {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***) //para gerar o ID

**private** **int** codigo;

@NotNull

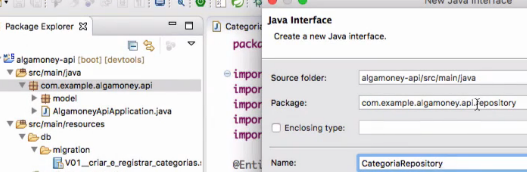
@Size(min=3, max=20)

**private** String nome;

..... gets e sets...

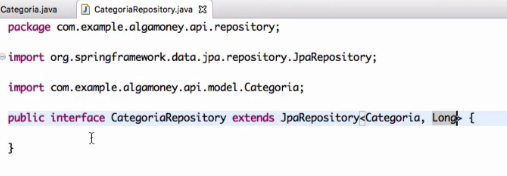
Gere também os métodos hashCode() e o equals()

Ainda na pasta com.example.algamoney.api crie o pacote **repository** e dentro dele a interface **CategoriaRepository**



**Figura 5** – Criando a interface CategoriaRepository

A interface CategoriaRepository deve extender a interface **JpaRepository**, informando a classe **Categoria** e o tipo de dado da chave primária de categoria.



Tipo de dado da chave primária de Categoria

A interface JpaRepository já trás todos os métodos básicos que deve ter um CRUD. E a implementação fica a cargo do **SpringDataJpa**, que já está no arquivo pom.xml.

Agora é necessário criar a classe que será o controlador dos recursos de Categoria.

Na pasta com.example.algamoney.api crie o pacote **resource** e dentro dele a classe **CategoriaResource**.

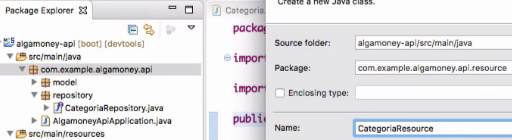


Figura 6 – criando a classe CategoriaResource.

Esta é a classe que vai expor tudo que esteja relacionado ao recurso Categoria. Ele é um controlador. Ele recebe requisições via http e o Spring localiza esta classe.

**@RestControle** é para indicar que esta classe é um controlador de recursos que recebe requests via http e retorna resultados em formato JSON.

**@RequestMapping("/categorias")** – quando no final da URL tiver "/categorias", o Spring buscará a classe **CategoriaResource**.

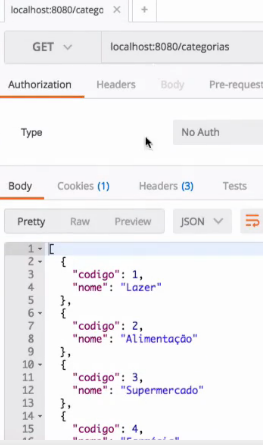
O método que contem o **@GetMapping** será o método a ser executado, a partir da URL de categorias, somente se a requisição estiver utilizando o recurso GET. Não poderá ter dois métodos com esta mesma marcação numa mesma classe.

A classe **CategoriaRepository** está sendo injetada aqui por meio da anotação **@Autowired**, trazendo todos os métodos CRUDs do SpringDataJpa para a classe CategoriaResource, sem ser necessário instanciar CategoriaRepository.



**Figura 7** – Classe CategoriaResource.

Após esta implementação já é possível testar o **resquest e o reponse** via URL, utilizando o POSTMAN. Inicie o servidor da aplicação e digite a url da aplicação no PostMan informando "/categoria" no final da URL.

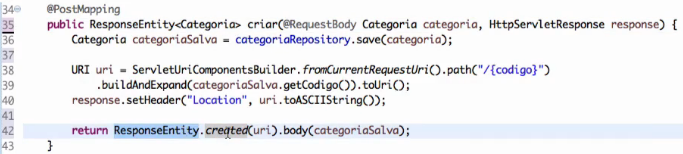
Figura 8 – Uso do PostMan para listar as categorias.

Exibição do resultado em formato JSON.

Utilizando o método GET

URL informada já com o "/categorias"

**10. Cadastrando categoria**



A notação **@PostMapping** é para receber dados via POST, logo quando o request for no modo POST e a URL terminar com "/categorias", o Spring executará o método criar().

A notação **@RequestBody**, incluída como parâmetro da função criar(), serve para o Spring enviar a resposta (request) em formato JSON.

Ao gravar a nova Categoria, o método save(), retorna para a variável categoriaSalva, o objeto Categoria, já com o código (ID) gerado, conforme está na linha 36. Na linha 38 ele está montando a URL que lista a nova categoria gravada, a partir do código dela já gerado, para depois redirecionar a página.

Para testar a gravação da nova categoria, utilize o POSTMAN, no modo POST, ao invés de GET. Na aba "Body do Postman, selecione a opção "raw" e o formato JSON.

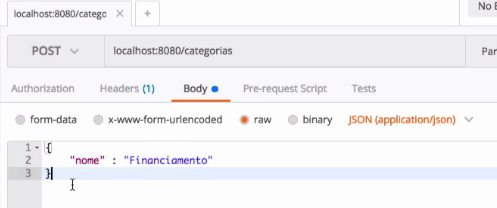
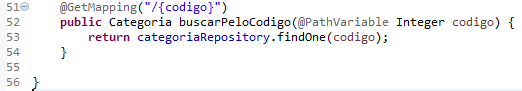


Figura 9 – uso do POSTMAN no modo "POST", para testar a gravação de uma nova categoria.

Observe que a URL é a mesma utilizada para listar as categorias (veja figura 8). A diferença é que agora estamos utilizando o modo POST ao invés de GET. É por isso que no método criar(), temos a notação @PostMapping.

Após executar o teste, observe que o POSTMAN informará no canto superior direito o Status "201 Created", indicando que conseguiu gravar o novo registro.

**11. Buscar categoria pelo código**



**@GetMapping("/{código}")** – se no final da URL for informado o código da categoria, o sistema executará o método "buscarPeloCodigo()". Ex.: localhost/categorias/7 (retornará a categoria cujo código é 7).

**@Pathvariable** – para o Spring pegar o código da categoria na URL.

**12. Validando atributos desconhecidos**

Por padrão, se for informado algum campo que não existe no BD, no request, para gravar um novo registro, a API do Spring simplesmente ignorará os campos que não existe. No entanto, pode-se implementar um tratamento para estes casos. Basta acrescentar a linha abaixo ao arquivo **application.properties**:

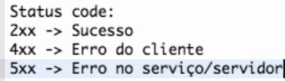
spring.jackson.deserialization.fail-on-unknown-properties=true

O **Jackson** é a ferramenta do Spring que transforma Objetos Java em JSON ou o inverso.

"**unknou-properties=true**" – se encontrar propriedades desconhecidas, exibir erro (fail).

O Spring retornará o código (Status code) de erro 400 Bad-request.

Exemplos de código retornados pelo Spring, a partir de respostas enviadas ao servidor:



**13. Tratando erros com ExceptionHandler**

O sistema deve estar preparado para exibir mensagens mais amigáveis para o usuário final e mensagens de erro mais detalhadas para o desenvolvedor.

Para tanto é necessário criar uma classe para tratamento de erros.

Na pasta principal, crie o pacote "exceptionhandler" e dentro dele a classe Alga

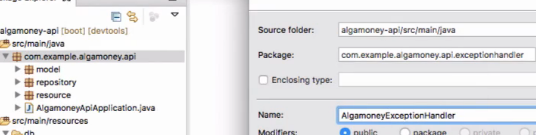


Figura 10 – criando a classe AlgamoneyExceptionHandler.

Esta é a classe. Ela deve extender a ResponseEntityExceptionHandler, que é a classe responsável por capturar exceções de respostas (response) das entidades (classes da camada model). Ela também funciona como um controlador, por isso a notação "**@ControllerAdvice**", que observa toda a aplicação.

@ControllerAdvice

**public** **class** AlgamoneyExceptionHandler **extends** ResponseEntityExceptionHandler {

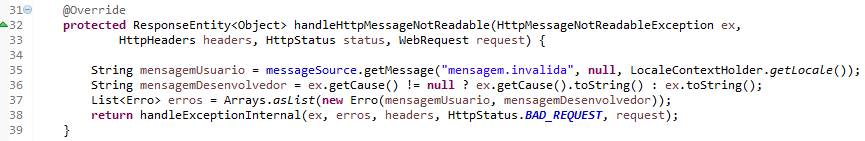
}

Esta classe utilizará mensagens customizadas, que deverão estar no arquivo "**messages.properties**", a ser criado na pasta "**scr/main/resources**". Uma das mensagens a ser escrita neste arquivo é a seguinte:

mensagem.invalida=Mensagem inv**\u00E1**lida

Observe que letras acentuadas deverão serem escritas no formato Unicode.

O primeiro método da classe AlgamoneyExceptionHandler será o seguinte:



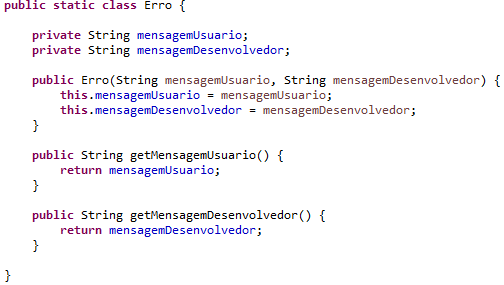
O método deverá ter exatamente este nome para que o Spring consiga identifica-lo.

Na linha 35 estamos utilizando a propriedade "**mensagem.invalida**", de forma que o Spring busca o texto da mensagem no arquivo "**messages.properties**", que é uma mensagem mais amigável para o usuário final.

Na 36 temos a mensagem a ser exibida ao desenvolvedor que possui um detalhamento do erro encontrado.

As mensagens são adicionadas a um array, que por sua vez, é passado como parâmetro para o método handleExceptionInternal(). (linha 38).

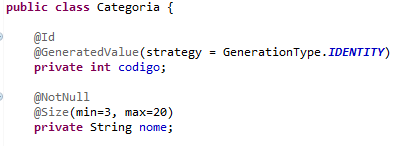
Na linha 37 foi utilizada uma classe estática chamada Erro, que está implementada dentro da própria classe AlgamoneyExceptionHandler, conforme abaixo:



As variáveis que recebem as mensagens de erro devem ser atributos de uma classe, no caso a classe Erro, para que possam ser convertidas para o formato JSON.

**14. Validando valores inválidos**

No arquivo pom.xml temos a dependência "spring-boot-starter-data-jpa". Esta dependência já incluir automaticamente o Hibernate Validation no nosso projeto, já configurado. Bastando então, apenas incluir a notação de campo obrigatório nos atributos da classe Categoria.



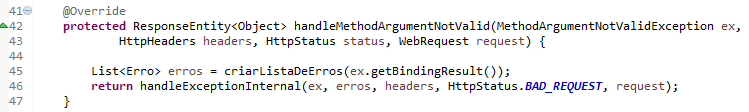
Também está sendo validado o tamanho máximo de caracteres no campo.

A notação @NotNull deverá ser importada da biblioteca "**javax.validation.constrains**".

Além disto, deverá ser acrescentada aos atributos do método "criar()", da classe **CategoriaResource**, a notação "**@Valid**", conforme abaixo:



Para tratar valores, é necessário criar o método abaixo na classe AlgamoneyExceptionHandler:

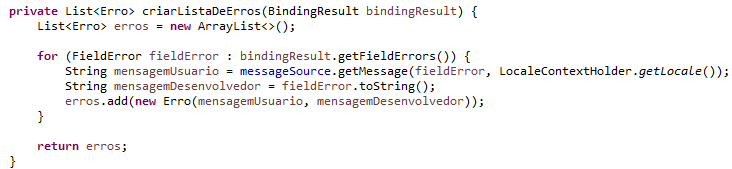


Considerando que o método acima, poderá validar vários campos de uma só vez, então é necessário criar uma lista que armazene todos os erros encontrados. Por isso, na linha 45, pode-se ver que estamos utilizando um método para gerar a lista de erros.

Se a tabela no BD, por exemplo, tiver mais de um campo obrigatório, a lista poderá armazenar um erro para cada campo obrigatório não preenchido.

Este método captura os campos ou atributos de Categoria, com erro.

Segue abaixo o método para gerar a lista de erros:



Entre as chaves será exibido o nome do atributo

As mensagens de erro relacionadas às classes "Entitys", serão tratadas no arquivo " ValidationMessages.properties", a ser criado exatamente com este nome, também na pasta "**scr/main/resources**".

Veja abaixo exemplo de mensagem a ser inserida neste arquivo:

javax.validation.constraints.NotNull.message=**{0}** **\u00e9** obrigat**\u00f3**rio(a)

javax.validation.constraints.Size.message=**{0}** deve ter o tamanho entre {min} e {max}

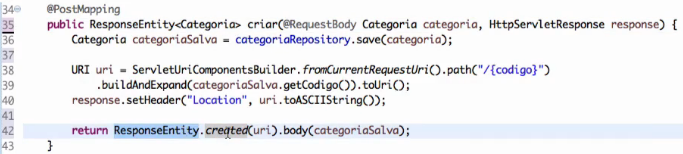
**15. Criar o cadastro de Pessoas**

O sistema terá um cadastro de pessoa. Com base nas instruções anteriores, já dá para criar o cadastro de pessoa com os campos: nome, endereco e ativo (boolean - se ativo ou não), conforme está no diagrama do sistema na figura 3.

**16. Trabalhando com o recurso Eventos do Spring**

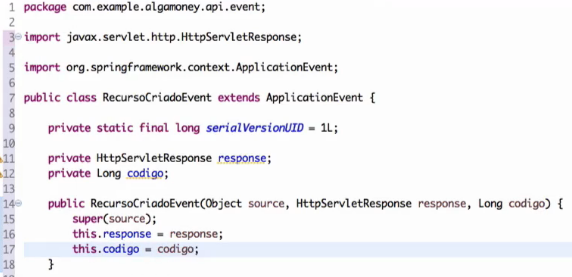
Eventos são acionados sempre que uma determinada ação é executada no código. Para que isso ocorra é necessário ter um "Listener", que fica observando o momento em que a ação determinada será executada, para que em seguida ele possa disparar (executar) o evento. Necessário então criar o evento e o listener desse evento.

No nosso sistema, utilizaremos este recurso nas classes CategoriaResource e PessoaResource, por estarem com trechos de código redundantes, nas linhas em que criam a URL de retorno.



Trecho de código redundante no método de gravação das classes Categoria e Pessoa.

O primeiro passo é criar a classe que vai representar o **evento**. Na pasta "com.example.algamoney.api" crie o pacote "event" e dentro dele a classe "RecursoCriadoEvent", conforme abaixo:



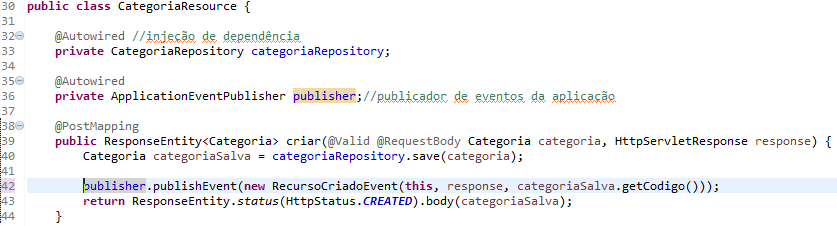
Gere somente os métodos GETs dos atributos codigo e response. Não é necessário os métodos SETs.

Em seguida criamos o Listener deste evento, conforme abaixo:



Observe que o código que está nas linhas 24 a 27 é o mesmo que está nas linhas 38 a 40 do método criar() da classe CategoriaResource. Portanto deveremos substituir este trecho de código das classes CategoriaResource e PessoaResource, por uma linha de comando que disparará o evento.

Na classe CategoriaResource, por exemplo, ficará da seguinte forma:



Observe que na linha 42 está sendo executado o evento RecursoCriadoEvent, que por sua vez, acionará automaticamente o listener RecursoCriadoListener. Para que isso ocorre é necessário utilizar a classe ApplicationEventPublisher. Ao publicar o evento, o listener consegue detectá-lo e assim poder executar o método correspondente da classe RecursoCriadoListener.

É necessário fazer a substituição do trecho de código também na classe PessoaResource.

**16. Removendo pessoas com DELETE**

Para testar no POSTMAN, faça o seguinte:

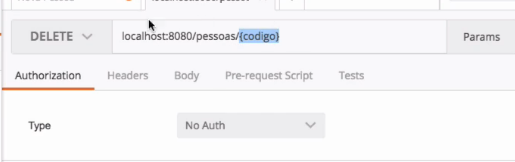
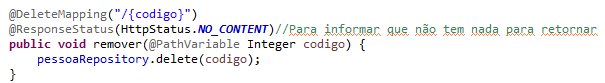


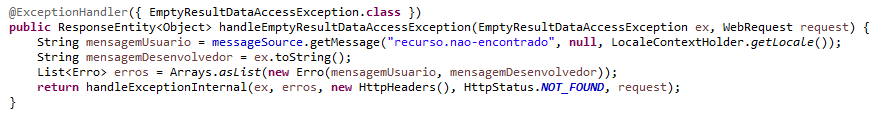
Figura 11 – Teste para remover uma pessa.

Se o POSTMAN retornar "**204 No Content**", quer dizer que deu certo.

Na classe PessoaResource, ficará da seguinte forma:



Para tratar valores inválidos na exclusão, é necessário acrescentar o método abaixo na classe AlgamoneyExceptionHandler:



A mensagem de retorno será "Recurso não encontrado". Veja que a mensagem está referindo a uma variável (contante), portanto a mensagem deverá ser configurada no arquivo "**Messages.properties**", conforme abaixo:

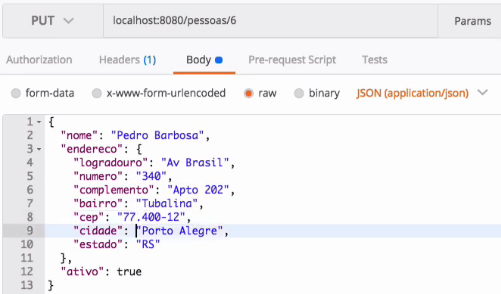
recurso.nao-encontrado=Recurso n\u00E3o econtrado

**17. Atualização de recursos REST**

Para alterar os dados de uma pessoa, por exemplo, pelo POSTMAN faremos um teste utilizando o recurso PUT (ao invés de GET ou POST). No caso de atualização via REST, mesmo que seja para alterar, por exemplo, somente o endereço de uma pessoa, é necessário enviar para o servidor todos os dados da pessoa, com apenas o endereço alterado. Uma forma de melhorar isso, é criar uma regra que verifique os valores preenchidos, de forma que, os campos que estiverem com valor NULL, não serão alterados, mas somente os que estiverem com os valores que realmente se deseja alterar. A desvantagem desta solução é que você não poderá mais alterar um campo para valor NULL. Uma outra solução é criar URL específicas para cada campo que se deseja alterar, que é uma solução mais utilizada para campos booleanos.

Na Figura 12, a seguir, temos um exemplo de teste no POSTMAN, alterando somente o endereço. Observe que para configurar o teste, foi selecionado a opção "raw" e o formato JSON. No body foi incluído o objeto a ser alterado. O código da pessoa é passado via URL (Ex.: localhost/pessoa/6) e o campo do código da pessoa **não deve constar no body**. Os campos que forem omitidos, receberão valor NULL na alteração.

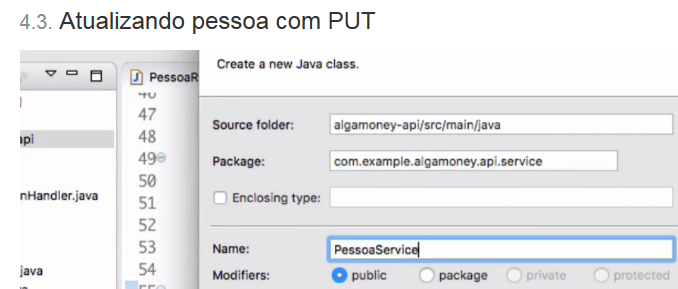
Nesse caso o código passado via URL, deve ser validado para saber se a pessoa realmente existe. Para tratar este código é necessário implementar um método de tratamento de excessão na classe AlgamoneyExceptionHandler. Poderemos utilizar o mesmo código de excessão que já implementamos para busca.



**Figura 12** – Atualizando endereço de pessoa. O código da pessoa não deve constar no body.

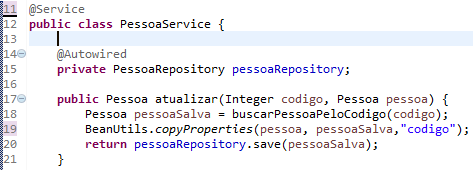
Regras de negócio nunca devem ficar na classe de controle, portanto é necessário implementar classes de serviço para tratamento e validação dos objetos (recursos). As classes de serviço são utilizadas quando é necessário fazer algum tratamento mais específico na operação a ser executada (CRUD). Portanto criaremos a classe de serviço para tratamento da alteração dos dados de pessoa, da seguinte forma:

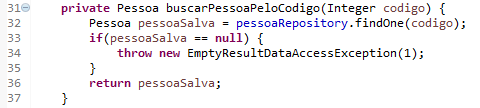
A classe deverá ser criada dentro do pacote "**service**"



**Figura 13** – Criando a classe PessoaService.

A classe ficará da seguinte forma:



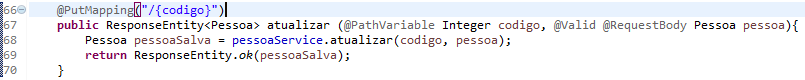


**@Service** - Esta anotação é para indicar que esta classe (PessoaService) é um componente do Spring que poderá ser injetada.

A linha 19, atualiza os dados de **pessoaSalva**. O terceiro parâmetro "**codigo**" é para indicar que o código da pessoa não deverá ser alterado em pessoaSalva.

Atenção para a importação da BeanUtils, - deverá ser da biblioteca do Spring (org.SpringFrameworks.beans).

Esta classe deverá ser injetada na classe "PessoaResource". O código em PessoaResource, deverá ficar da seguinte forma:



**@PathVariable** é para capturar da URL, o código da pessoa.

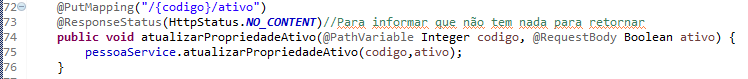
**@Valid @RequestBody** – para validar os dados de pessoa (campos obrigatórios).

A linha 68 está utilizando o método "atualizar()" da classe PessoaService, que foi injetada na classe PessoaResource, por meio da linha de comando abaixo, implementada nas linhas iniciais desta classe:



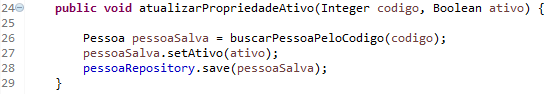
**18. Atualizando somente o campo Boolean Ativo**

O código em PessoaResource, deverá ficar da seguinte forma:

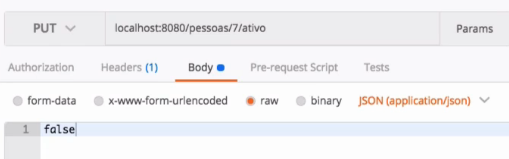


Observe que na notação "@PutMapping" está indicando que na URL deverá constar o código da pessoa e o nome do campo booleano, no caso o campo "ativo".

Na classe **PessoaService**, o método **atualizarPropriedadeAtivo**(), ficará da seguinte forma:



Para testar no POSTMAN, faça da seguinte forma, no caso de alteração da propriedade "ativo" da pessoa de código igual a 7:



**Figura 14** – Alterando somente a propriedade ativo.

Execute o teste passando valores inválidos ou nulo e verifique o tipo de mensagem de erro retornada. Se nenhum valor for informado, ele assume o valor default, que é TRUE.

**19. Criando a entidade Lançamento**

Esta entidade possui algumas características que necessita de implementações diferentes de Pessoa e Categoria.

Ela possui dois campos de data e utiliza constantes para diferenciar lançamentos de Débito e de Crédito.

Script para criação da tabela a ser implementado no arquivo de migração:

CREATE TABLE public.lancamento

(

codigo serial NOT NULL,

descricao VARCHAR(50) NOT NULL,

data\_vencimento date NOT NULL,

data\_pagamento date,

valor numeric(10,2) NOT NULL,

observacao VARCHAR(100),

tipo VARCHAR(20) NOT NULL,

codigo\_categoria integer NOT NULL,

codigo\_pessoa integer NOT NULL,

CONSTRAINT codigolancamento\_pkey PRIMARY KEY (codigo),

CONSTRAINT fk\_categoria FOREIGN KEY (codigo\_categoria)

REFERENCES categoria (codigo),

CONSTRAINT fk\_pessoa FOREIGN KEY (codigo\_pessoa)

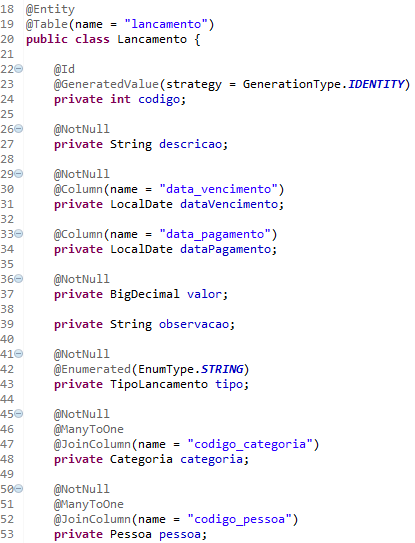
REFERENCES pessoa (codigo)

);

Exemplo do Scritp para inclusão de registro:

INSERT INTO public.lancamento (descricao, data\_vencimento, data\_pagamento, valor, observacao, tipo, codigo\_categoria, codigo\_pessoa) values ('Salário mensal', '2017-06-10', null, 6500.00, 'Distribuição de lucros', 'RECEITA', 1, 1);

A classe **Lancamento** no pacote **Model**, ficará da seguinte forma:

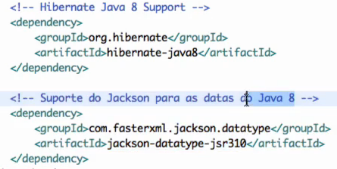


Para indicar que TipoLancamento é um enum.

LocalDate é da biblioteca do java.time.LocalDate

Para implementar a lista de lançamentos e a busca de lançamento por código, tome como exemplo os códigos anteriores.

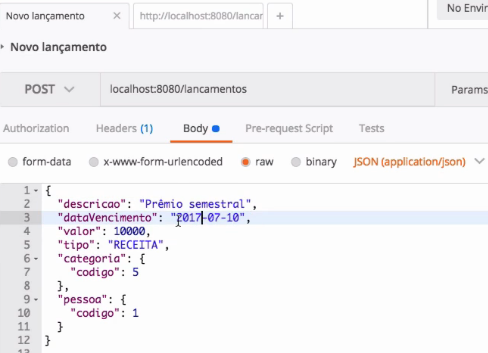
Necessário acrescentar mais duas dependências no arquivo **pom.xml**, para tratamento de datas:



No arquivo **application.properties**, é necessário acrescentar a linha de código abaixo, para apresentação da data no formato padrão internacional:

spring.jackson.date-format=yyyy-MM-dd

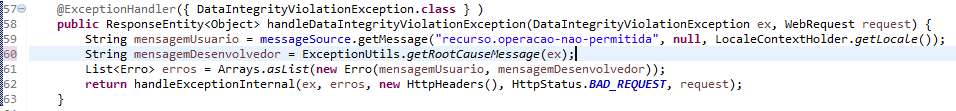
Para efetuar os testes de cadastro de lançamento no POSTMAN, faça da seguinte forma:



**Figura 15** – Cadastrando um lançamento.

**20. Validando Lançamento**

No caso do usuário informar uma categoria inexistente, o sistema retornará uma mensagem de inconsistência de dados. Para tratar esta inconsistência, necessário implementá-la na classe AlgamoneyExceptionHandler.

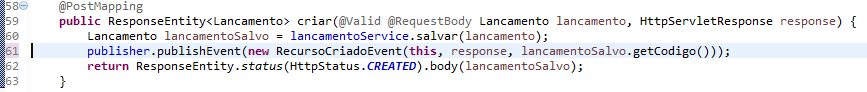


Para a linha 59, é necessário implementar a mensagem no arquivo **application.properties**.

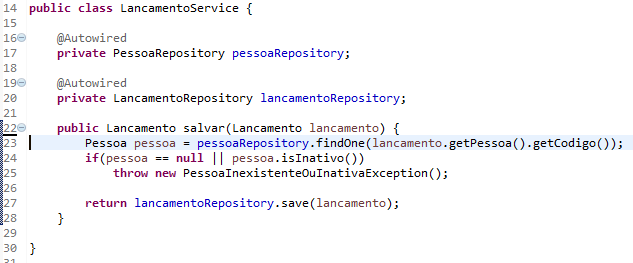
Na linha 60, foi utilizada a LIB commons-lang3, para melhor detalhamento de mensagens de erro e no arquivo **pom.xml**, inclua a seguinte dependência:



O método para cadastrar um lançamento da classe **LancamentoSource**, ficou da seguinte forma:



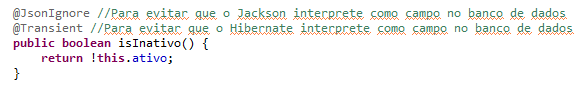
E na classe LancamentoService, assim:



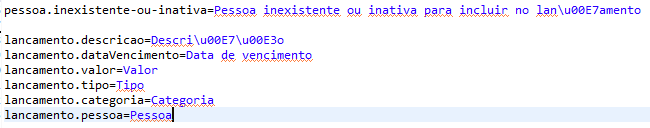
Para não aceitar o cadastro de lançamento para pessoas inativas.

Na **linha 23**, utilizasse a busca da pessoa da interface PessoaRepository, ao invés do método da classe PessoaService, pelo fato da mesma retornar um erro não apropriado para este fim.

Na **classe Pessoa**, o método **isInativo( )**, deverá ficar da seguinte forma:



Para melhorar a exibição de mensagens de campos obrigatórios de Lançamento, incluir as seguintes linhas no arquivo **messages.properties**:



**21. Implementando pesquisa de Lançamento utilizando recursos do JPA**

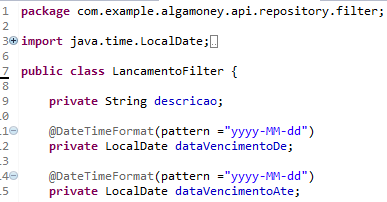
A ideia é criar uma classe somente para os filtros de consulta em Lançamento. Esta classe deverá ser passada como parâmetro no método de consulta.

Na classe LancamentoResource, renomeie o método listar() para pesquisar() e acrescente a classe LancamentoFilter como parâmetro deste método:



O método filtrar(), será implementado mais adiante.

Crie a classe **LancamentoFilter** dentro do pacote "**repository.filter**":

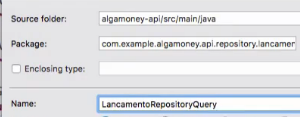


O filtro será por descrição ou data de vencimento, sendo que, para data de vencimento teremos o intervalo "De" e "Ate".

Para as datas, necessário a notação @DateTimeFormat(pattern = "yyy-MM-dd")

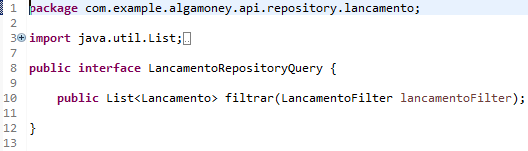
Gere os GETs e SETs para estes atributos.

O próximo passo é criar uma interface. O nome dela deverá ser **LancamentoRepositoryQuery** e deverá ser criada no pacote "**repository.lancamento**":



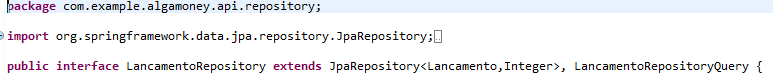
**Figura 16** – Implementando o filtro para lançamento.

Nesta interface inclua o método filtrar, conforme abaixo, de modo que ela possa ser implementado na classe que extende esta interface.



O nome da interface tem que ser exatamente este para que o Spring possa reconhecer como um filtro para lançamento.

A interface **LancamentoRepository** deve extender esta nova interface (LancamentoRepositoryQuery)

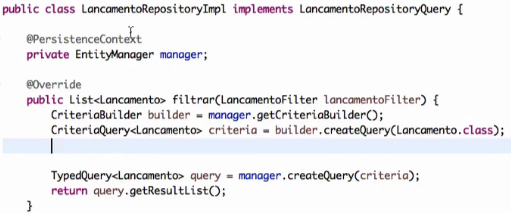


Agora é necessário implementar a consulta que utilizará os filtros.

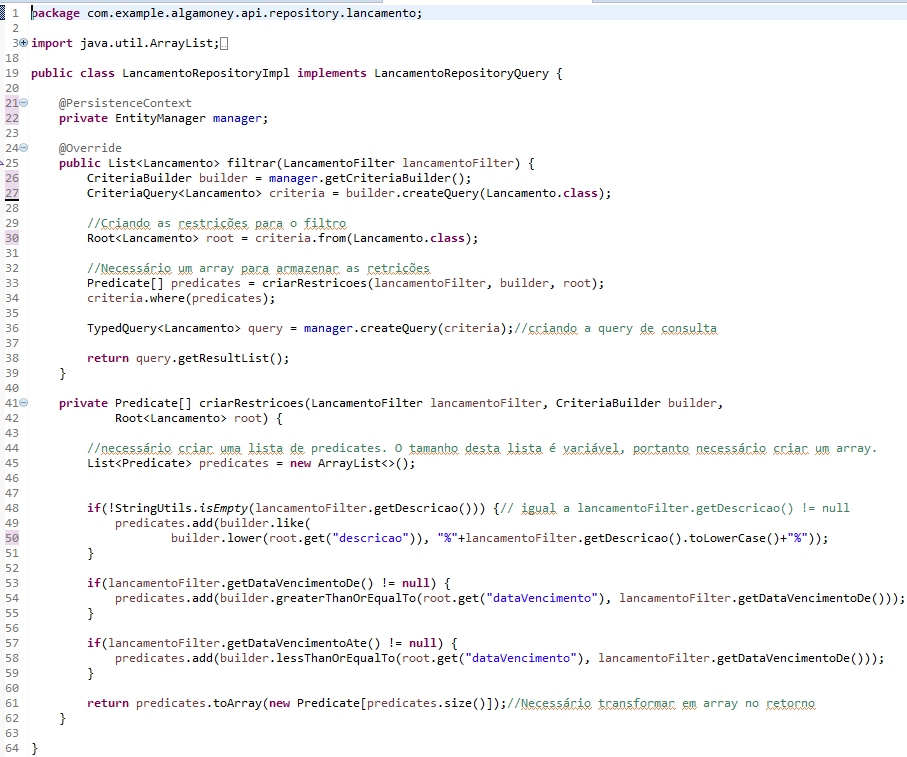
Crie a classe LancamentoRepositoryImpl com Implements LancamentoRepositoryQuery, conforme está na página seguinte.

Nessa implementação foi utilizado o "Criteria" do JPA, pelo motivo do "Criteria" do Hibernate já estar depreciado.

Para implementar a Criteria **sem os filtros**, basicamente seria o seguinte código:

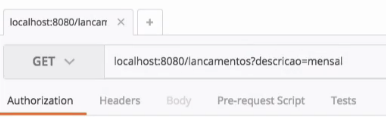


A seguir, temos a implementação já com os filtros de consulta:



No curso é demonstrado ainda a utilização de "**MetalModels**" para não ser necessário informar o nome dos campos da tabela no formato String (entre aspas), conforme está nas linhas 50, 54 e 58. É necessário criar uma nova classe e adicionar a dependência "hibernate-jpamodelgen". Optei em não implementar por ser mais simples.

Para testar no POSTMAN, basta o seguinte:



**Figura 17** – Testando o filtro de lançamento no POSTMAN.

Ele listará todos os lançamentos que contenham o termo "mensal" na descrição.

Uma pesquisa por período de data de vencimento ficaria da seguinte forma:



**Figura 18** – Pesquisa por períodos de data de vencimento.

**22. Implementando a paginação de lançamentos**

No método pesquisar(), da classe **LancamentoResource**, é necessário acrescentar mais um parâmetro. Acrescente a ele o parâmetro "Pageable" da biblioteca **org.springframework.data.domain**.

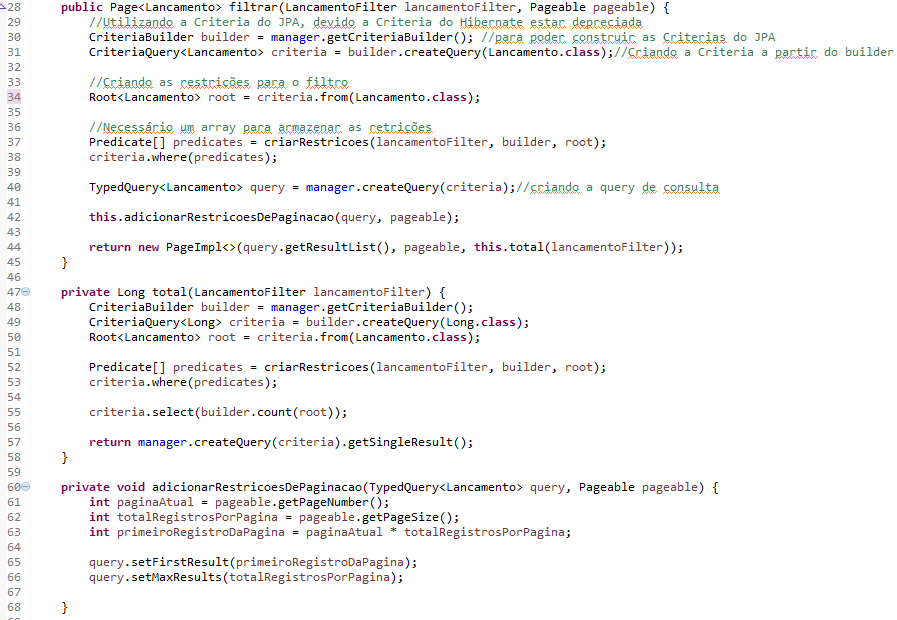


Este método (pesquisar) terá agora que retornar uma paginação. Ao invés de List<Lancamento>, deverá ser Page<Lancamento>. O Page também é da biblioteca **org.springframework.data.domain**.



O método **lancamentoRepository.filtrar()**, também deverá receber o pageable como parâmetro.

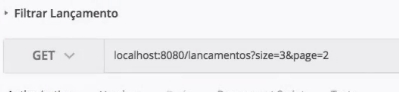
Agora é necessário fazer estas alterações também na interface **LancamentoRepositoryQuery** e no método **filtrar()** da classe **LancamentoRepositoryImpl**. Este método será modificado para implementar a paginação, conforme código abaixo:



* Linha 28 – o método está retornando agora um Page<Lancamento> e recebendo como parâmetro um Pageable.
* Linha 42 – criamos o método "adicionarRestricoesDePaginacao(), que implementa a forma como funcionará a paginação.
* Linha 44 – retorna um new **PageImpl<>()**, que também é da biblioteca **org.springframework.data.domain**. Como parâmetro, além do ResultList e do pageable, ele recebe o método total(), que é o método que calcula o total de registro que será retornado como resultado da consulta por filtro.
* Linha 66 - seta o máximo de registros por página. Este valor será informado via URL. Exemplo: "size=20", para vinte registros por página.

Estas informações serão exibidas no final do resultado que é retornado nos testes efetuados no POSTMAN.

No exemplo abaixo, o POSTMAN retornará 3 registros por página a partir da página 2:



**Figura 19** – Listando 3 registros por página a partir da página 2.

Ao final da lista de lançamentos, o POSTMAN exibirá as seguintes informações:

"Last" está como "false" indicando que não é a última página.

"totalElements" é o número total de registros recuperados do BD.

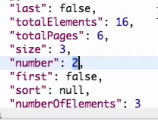
"totalPages" – total de páginas.

"size" – quantidade de registros por página.

"number" – número da página atual.

"first" – está com o valor "false", indicando que não é a primeira página.

"numberOfElements" – quantidade de registros mostrados. Se no BD tivesse menos que três elementos cadastrados, este item exibiria um valor menor que 3.

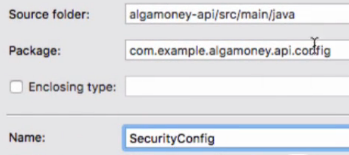


**23. Implementando autenticação Basic (acesso com login e senha)**

O primeiro passo é adicionar a dependência SpringSecurity no arquivo pom.xml



Em seguida, crie a classe de configuração da autenticação, no pacote "**config**", com o nome de "**SecurityConfig"**:



**Figura 20** – Criando a classe de configuração da autenticação Basic.

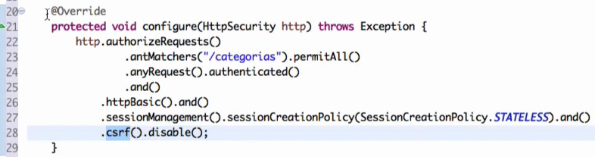
A classe SecurityConfig deverá extender a classe WebSecurityConfigurerAdapter e receber as notações @Configuration e @EnableWebSecurity, para ser reconhecida pelo Spring:



Simulação do login e senha do usuário que está utilizando o sistema.

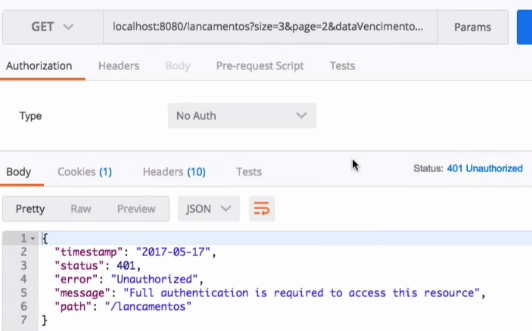
Aqui está sendo demonstrado o login e a senha sendo passado direto pelo código. Não estamos ainda implementando o Role.

Acrescente a esta classe o seguinte método:



* linha 23 – está permitindo acesso sem senha somente para a lista de categorias
* Linha 26 – está informando que a implementação é do tipo httpBasic. Mais adiante veremos outra forma de implementação de autenticação mais segura que a Basic.
* Linha 27 – implementa a não utilização de Sessão para não manter o login e senha na sessão.
* Linha 28 – para evitar acessos via JavaScript sem autenticação.

Primeiro faça os testes no POSTMAN sem informar login e senha, para ver a mensagem de retorno.



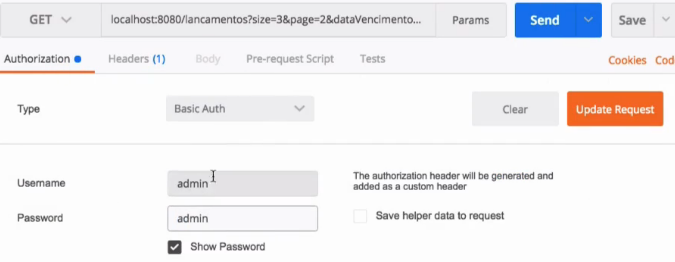
**Figura 21** – Mensagens retornadas pelo POSTMAN ao tentar acessar sem login e senha.

Observe que o status retornado foi "**401 Unauthorized**".

Se acessar a lista de categoria não será necessário informar login e senha.

Para informar o login e senha é necessário clicar na aba "**Authorization**" e no menu Type, selecionar a opção "**Basic Auth**".

Execute uma consulta em Lançamentos.



**Figura 22** – Informando o login e a senha.

Após informar o login e senha, clique no botão "**Update Request**". Para que estas informações passem a fazer parte do Header da URL. Em seguida clique no botão **Send**.

**24. Implementando segurança com OAuth 2**

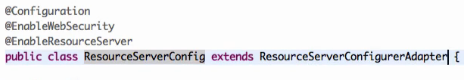
Primeiro passo é incluir no arquivo pom.xml a dependência do OAuth 2.



Segundo passo – renomeie a classe **SecurityConfig**, para **ResourceServerConfig**.

Esta classe deverá passar a extender a classe **ResourceServerConfigureAdapter**.

Acrescente a ela a notação "**@EnableResourceServer**", conforme abaixo:

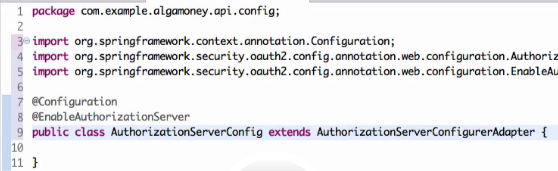


No método "configure()" altere a notação "**@Override**" para "**@Autowired**". E altere a visibilidade para "**public**" no segundo método "condigure()".



Neste segundo método "configure()" exclua a linha do "**httpBasic().add()**".

No pacote api.config, crie a classe "**AuthorizationServerConfig**", que deverá extender a classe "**AuthorizationServerConfigAdapter**", incluindo as notações conforme código que se segue:



O primeiro método "configure()" desta classe, deverá ficar da seguinte forma:



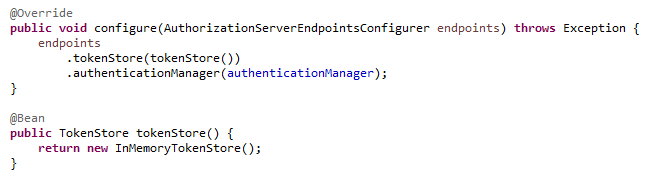
Este é o nome que daremos ao nosso front-end.

As informações de acesso ficaram armazenados na memória do computador, mas também poderia ser armazenada em um banco de dados.

Tempo máximo de duração do acesso 30 min.

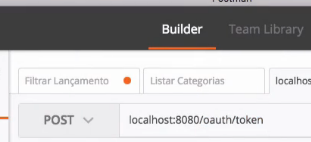
Esta é a configuração de login e perfil de acesso do nosso front-end ao nosso back-end.

Acrescente a esta classe, mais dois métodos:



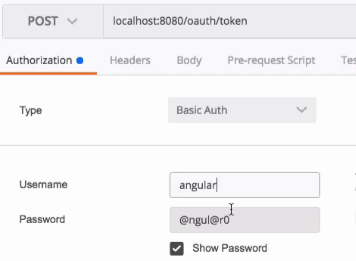
Armazena o Token em memória, para que possa ser recuperado quando necessário e verificar se ainda é um Token válido. Posteriormente será substituído por outro tipo de Token.

Para testar este tipo de autenticação no POSTMAN é necessário criar uma nova requisição do tipo POST e utilizar a URL "**localhost/oauth/token**".



**Figura 23** – Configurando o teste de autenticação para OAuth 2.

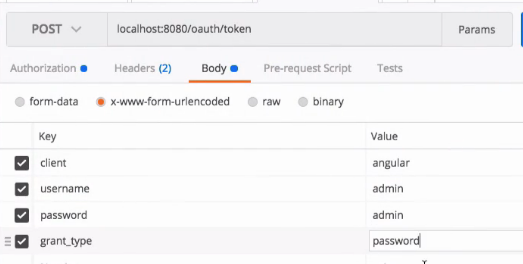
O usuário e senha a ser informado agora são os que foram implementados no método "configure()", da classe **AuthorizationServerConfig**, conforme imagem a seguir:



**Figura 24** – Configurando o login e senha do teste de autenticação para OAuth 2.

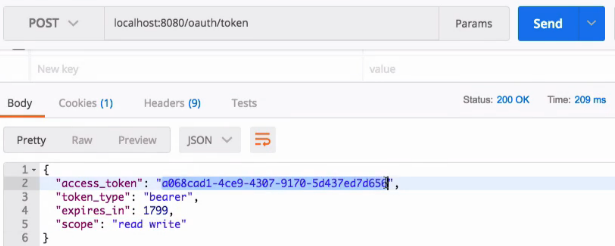
Não esqueça de clicar no botão **UpdateRequest**.

Além desta configuração ainda é necessário acrescentar mais informações de autenticação no body do POSTMAN. Clique na aba **Body**, selecione a opção **x-www-form-urlencoded** e acrescente as informações conforme demonstrado na figura abaixo:



**Figura 25** – Acrescentando informações no Body, para configuração do teste de autenticação para OAuth 2.

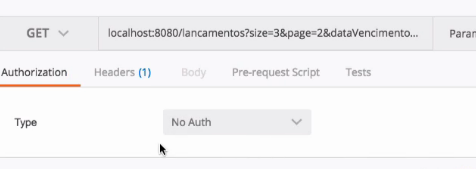
Ao clicar no botão **Send**, do POSTMAN, ele gerará a seguinte informação:



**Figura 26** – Informações geradas após clicar no botão **Send**, do teste de autenticação para OAuth 2.

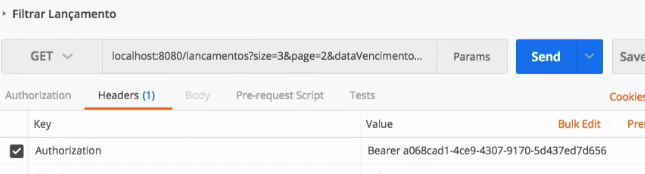
Mostra o tipo do token o tempo que ele levará para expirar e o seu escopo.

Copie o código do **access\_token** que foi gerado pelo POSTMAN. Agora na aba **Authorization**, retire a configuração do Basic Auth, conforme abaixo:



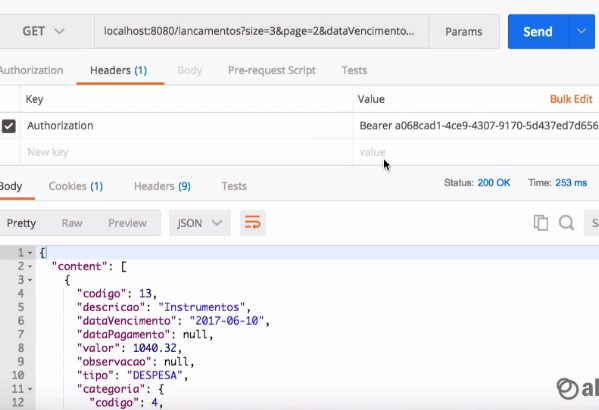
**Figura 27** – Desabilitar a autenticação básica.

Agora, na aba **Headers** de lançamentos, o valor de Authorization será "**Bearer**" seguido de um espaço em branco e do código do **access\_token**, que você copiou da tela anterior (figura 26). Lembrando que o termo "**Bearer**" é o que foi exibido na tela da figura 26 também, no parâmetro **token\_type**.



**Figura 28** – Configurando os parâmetros de validação do Token, no Header.

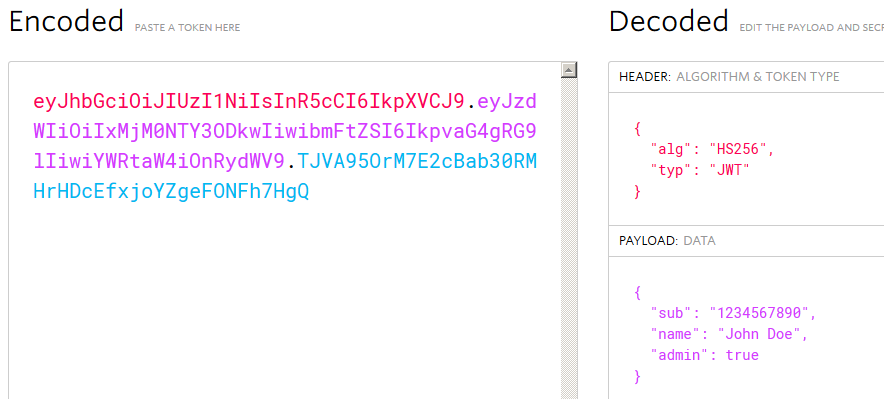
Agora é possível acessar/visualizar os dados do lançamento, que estão protegidos por senha. Basta informar a url de lançamento e clicar no botão **Send**.



**Figura 29** – Exibindo os dados de lançamento utilizando a autenticação OAuth 2.

**25. JSON Web Tokens – JWT (https://jwt.io)**

Este é um Token mais completo, com muito mais informações dentro dele. Neste tipo de Token é possível armazenar dados no formato JSON, que podem serem decodificados quando necessário. O próprio Angular pode fazer isso. Veja exemplo:

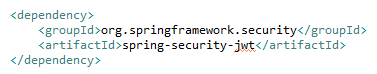


**Figura 30** – Exemplo de informações armazenadas em um Web Token.

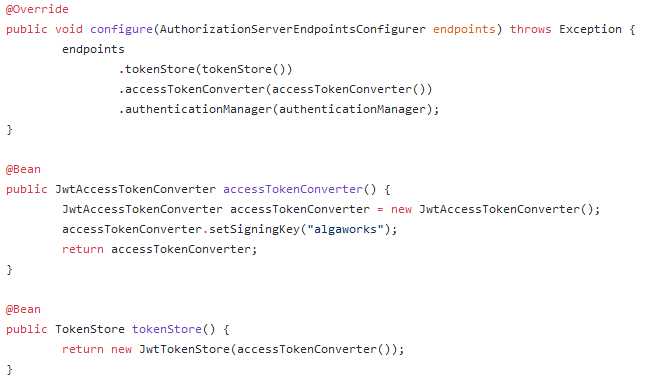
Em um web token pode-se armazenar o nome do usuário, o perfil e etc. Recomenda-se limitar o volume de informações enviadas via token, de modo a não deixar a string em tamanho muito grande.

A senha do usuário também pode ser armazenada no token, mas não decodificada.

Para implementar o web token é necessário adicionar mais uma dependência ao pom.xml

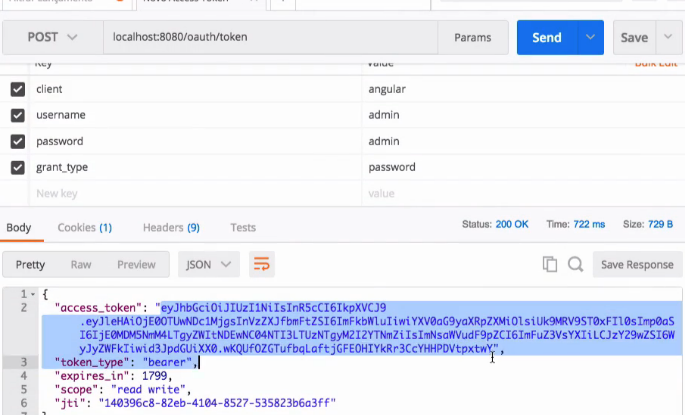


As alterações a serem feitas nas classes de autenticação serão mínimas. Necessário acrescentar um **conversor de token** no **endpoint** que está no método "configure()", da classe **AuthorizationServerConfig**.



O token não é mais armazenado em memória, mas sim em um JwTokenStore()

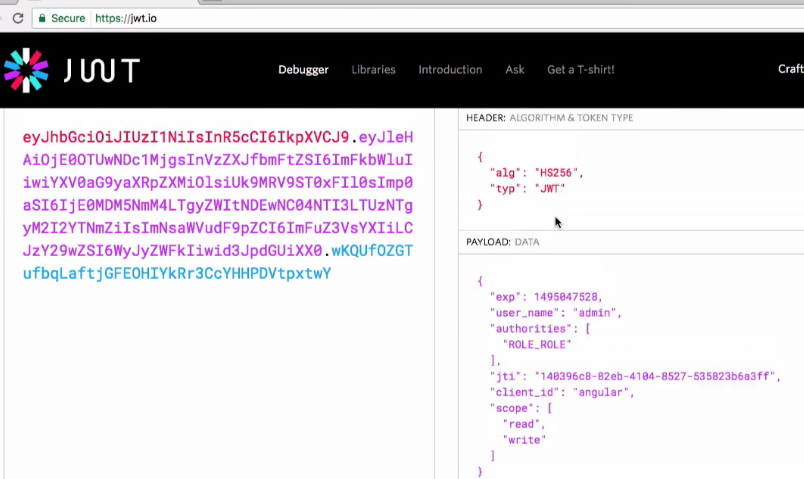
O teste no POSTMAN é o mesmo do anterior. A diferença será no novo código do Token que será gerado, que terá agora um tamanho bem maior.



**Figura 31** – Exibindo os dados do novo token na autenticação OAuth 2, com Jason Web Token.

Substitua o token que colocamos no header de lançamentos, por este novo token e execute o teste do POSTMAN.

Colando este novo token no site da jwt.io, é possível visualizar as informações armazenadas nele, conforme mostra a figura abaixo:



**Figura 32** – Exibindo os dados do novo token no site jwt.io.

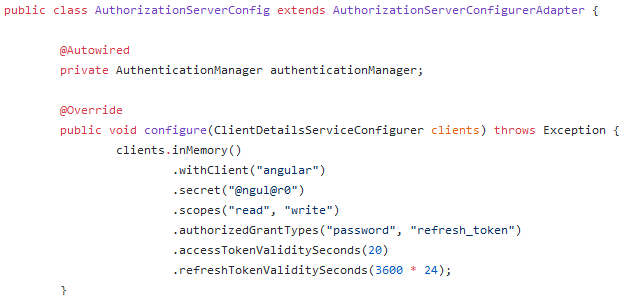
**26. Renovando o access token com o refresh token**

Na configuração anterior, a duração do token está limitada a 30 minutos. Expirando este prazo será necessário gerar um novo token. Caso contrário, o usuário será redirecionado para a tela de login da aplicação.

Estando o usuário conectado e em atividade, será necessário implementar então o **refresh token**, de modo a evitar a expiração do token.

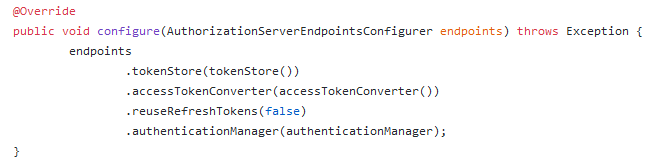
A expiração do token deve ter um tempo reduzido como forma de garantir maior segurança a uma aplicação. Além disso este token deve ser armazenado em um cookie do Browser, e a cada renovação do token, o browser deverá atualizá-lo no cookie, aumentando assim a segurança da aplicação. Mais adiante veremos como implementar o armazenamento em cookie.

O primeiro passo para implementar o refresh do token é acrescentar um novo tipo de fluxo de autenticação no método configure() da classe AuthorizationServerConfig, conforme abaixo:

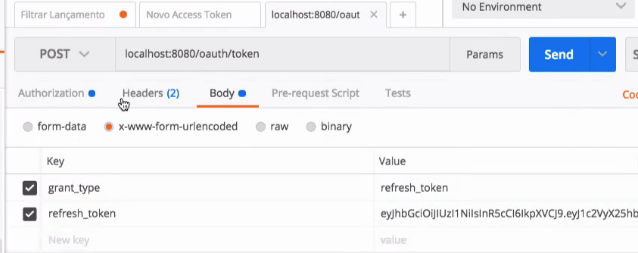


Acrescentando o "refresh\_token"

Configuramos também o refresh\_token, com tempo de duração de **24 horas**. Dessa forma garantimos que enquanto o usuário estiver logado, novos tokens serão gerados. Além do mais é necessário configurar no **endpoint** a **não reutilização do refresh\_token após 24 horas**, como medida de segurança, conforme abaixo.



Os testes do POSTMAN utilizando o **refresh\_token** é semelhante ao teste anterior. Muda somente os parâmetros do Body, que deverá ser informado o código do refresh\_token gerado.



Ao executar o teste da figura 21, o POSTMAN exibirá, além do código do token, também o código do refresh\_token.

**Figura 33** – Teste com refresh\_token.

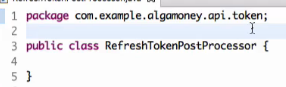
O novo token gerado com o refresh\_token, deverá ser copiado e colado no teste de lançamentos, para que se consiga visualizar os registros recuperados do banco.

A ideia é fazer a implementação de uma função que busque o refresh\_token sempre que o token expirar. A partir do refresh\_token é possível gerar um novo token. Dessa forma evita-se que o sistema fique pedindo login sempre que o token expirar. Essa implementação será feita no **Angular**. No entanto, o Angular só terá acesso ao refresh\_token se ele estiver armazenado em um cookie.

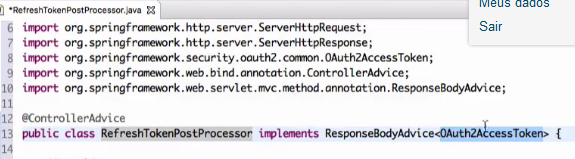
**27. Movendo o refresh token para o cookie**

Deverá ser utilizada a interface chamada **ResponseBoryAdvice**, para capturar o refresh\_token no momento em que ele for requisitado e em seguida, gravá-lo em um cookie. Antes mesmo de ser exibido no teste do POSTMAN, o refresh\_token será retirado da requisição e gravado em um cookie.

O primeiro passo é criar uma classe, no pacote api.token, com o nome de **RefreshTokenPostProcessor**.



Esta classe deverá ter a notação **@ControllerAdvice** e implementar a interface **ResponseBodyAdvice** para o objeto **OAuth2AccessToken**, conforme abaixo:



Este é o objeto que o POSTMAN exibe quando executamos o teste localhost/oauth/token. É ele que deverá ser capturado.

Esta classe deverá implementar os métodos da forma como se segue:



Para o cookie expirar em 30 dias.

Utilizar https somente na produção.

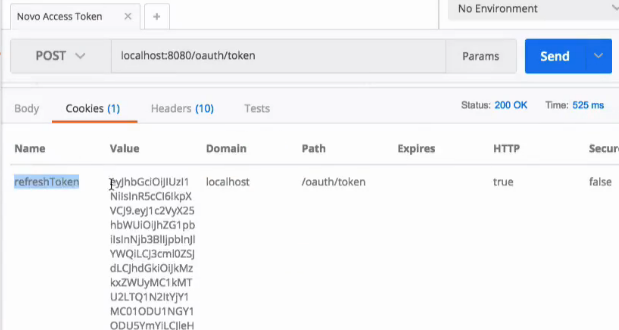
Nas linhas 32 e 33 foi necessário converter o request e o response para HttpServletRequest/ HttpServletResponse, para que fosse possível criar o cookie que armazenará o refresh\_token.

A linha 37 captura o token da requisição.

A linha 38 executa a função que cria o cookie e adiciona o refresh\_token.

A linha 39 remove o refresh\_token da requisição, por questão de segurança.

Agora no teste com o POSTMAN é possível visualizar o refresh\_token armazenado no cookie e não mais sendo exibido pela url localhost/oauth/token. Para visualizá-lo clique na aba **Cookies**.



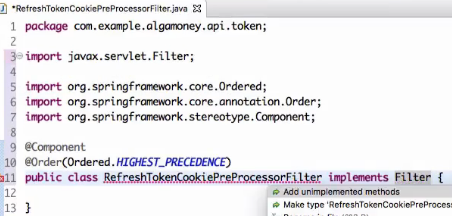
**Figura 34** – visualizando o código do refresh\_token no cookie que foi gerado. Veja que está somente em HTTP e não em HTTPS, por enquanto.

Estando o refresh\_token no cookie é necessário agora implementar o código que captura o refresh\_token a partir do cookie.

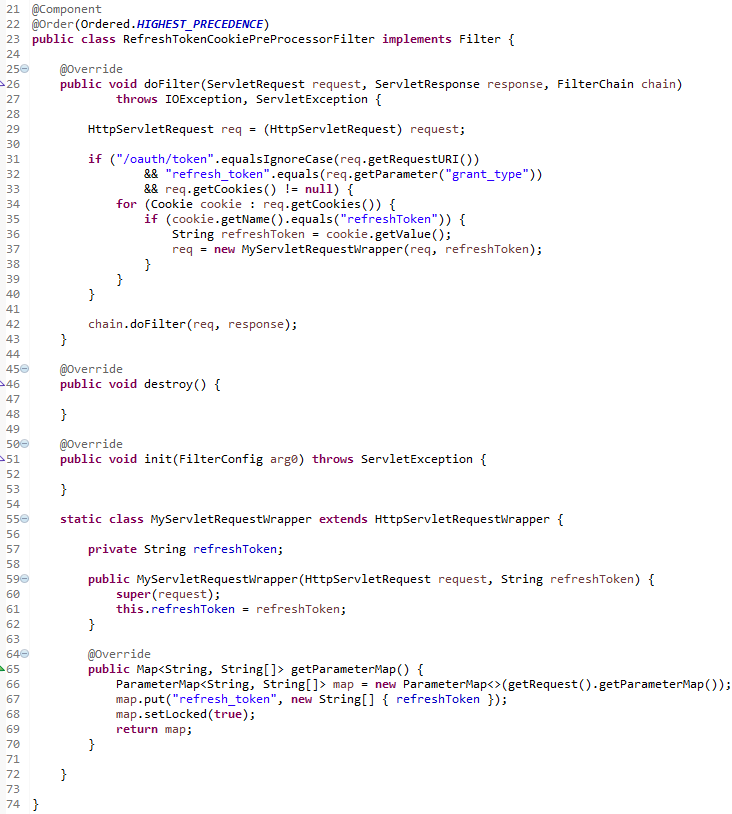
**28. Movendo o refresh token do cookie para a requisição**

O conteúdo de uma requisição não poderá ser alterado, portanto a solução é criar uma nova requisição via código, e adicionar o refresh\_token a ela.

O primeiro passo é criar a classe (Filtro) que buscará o refresh\_token no cookie. Ela deverá ter o nome **RefreshTokenCookiePreProcessorFilter** como abaixo:



A notação **$order(Ordered.HIGHEST\_PRECEDENCE)** serve para dar prioridade de processamento desta classe antes de qualquer requisição. Ela implementa o Filter do **javax.servlet.Filter**.



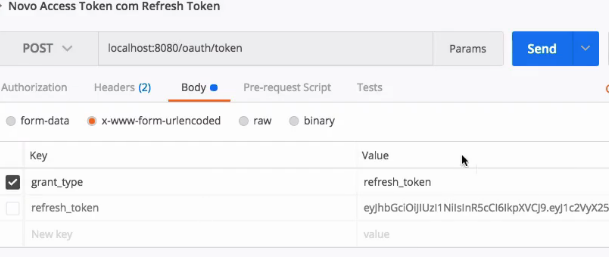
Necessário criar uma nova requisição para armazenar o refresh\_token, uma vez que não tem como alterar a requisição atual. A requisição atual será substituída pela nova requisição.

Parâmetros da nova requisição, já com o refresh\_token.

Capturando o refresh\_token do Cookie.

Pegar valor do cookie, somente se tiver um refresh\_token.

Agora o refresh\_token já pode ser retirado do body da requisição, no teste a ser efetuado no POSTMAN. Observe que agora ele está desmarcado no body.



**Figura 35** – visualizando o código do refresh\_token sem passá-lo via body, mas sim recuperando-o do cookie, via código.

Para testar o lançamento é necessário copiar o novo token gerado e colá-lo no header de lançamento.

Recomenda-se manter a expiração do token com tempo mínimo, para garantir maior segurança. O refresh\_token, armazenado no cookie gerará novo token sempre que necessário.

**29 – CORS – Requisição HTTP em Origens cruzadas (diferentes)**

Os browsers, por questão de segurança, não permitem que requisições HTTP sejam feitas a servidores diferentes do servidor de origem (servidor de origem da requisição). Por exemplo, ter em uma página HTML, um código javaScript que busca dados de outros servidores. Muitas aplicações precisam utilizar este tipo de recurso, mas para conseguirem implementa-las, será necessário habilitar o CORs, para então adicionar ao código as exceções, ou seja, os servidores externos que poderão serem acessados.

Vejamos por exemplo na classe CategoriaResources, método listar(), que lista as categorias.

A notação @CrossOrigin do SpringBoot, permitirá receber requisições do servidor localhost:8000, sendo que a aplicação estará rodando em localhost:8080.

Além do mais, as requisições deverão serem feitas em intervalos de 10 minutos.

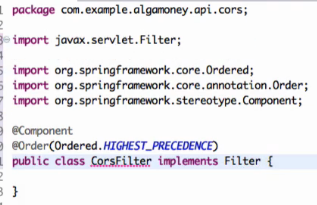


A notação @CrossOrigin também pode ser feita na Classe, de modo a permitir acesso a todos os seus métodos. De outra forma, também é possível colocar esta permissão para toda a aplicação (configuração global), não somente a uma classe.

No projeto deste curso a habilitação do CORS não será implementada utilizando notação (anotations), devido a incompatibilidade com o SpringSecurity ainda nesta versão. Para solucionar este problema, a habilitação será feita via Filtro.

**30. Criando Filtro para CORS.**

Criar uma classe com o nome "CorsFilter", no pacote "cors". Acrescentar a notação "@Component" para ser identificada como um componente do SpringBoot e a notação "@Order..." para dar prioridade de processamento.



Os métodos desta classe ficarão da seguinte forma:



Na linha 34 ele verifica se a requisição vem de um servidor de origem diferente. Nesse caso ela é inicialmente do tipo "OPTIONS". Se for, as linhas seguintes define os tipos de acesso que a requisição deverá ter para habilitar o CORS e não ser bloqueada pelo SpringSecutiry.

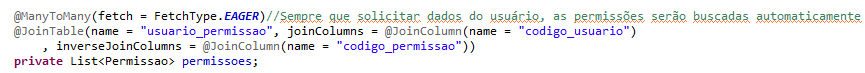
Na linha 34 ele ainda valida a origem da requisição que o acesso será permitido.

Na linha 35 ele habilita requisições do tipo POST, GET....

**31. Criar a tabela usuário e retirar do código o usuário e senha que estavam já definidos direto no código**

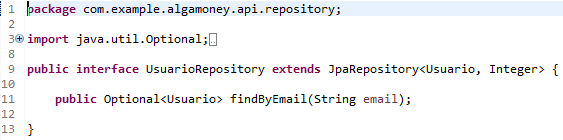
Criar o arquivo da migração do usuário, que gerará as tabelas "usuario", "permissao" e "usuario\_permissao". Acrescentar também os scripts de inserção de registros nestas tabelas. A tabela "usuário" deverá ter os campos nome, email e senha, o e-mail será utilizado para autenticação.

Criar a classe model para as entidades "Usuario" e "Permissao". A classe "Usuario", além dos atributos nome, email e senha, terá o atributo "permissoes", que deverá estar implementando com as anotações de mapeamento do relacionamento entre usuário e permissões, da seguinte forma:



Não é necessário mapeamento de relacionamento na classe "Permissao".

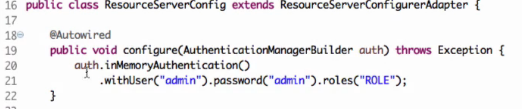
Próximo passo é criar a interface **UsuarioRepository**, conforme abaixo:



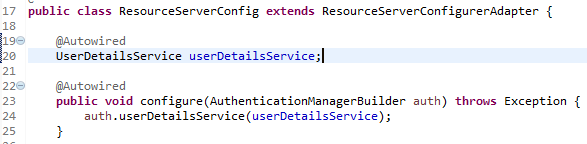
Na linha 11, foi utilizado o recurso Optional, que é uma alternativa para retornar NULL, caso não encontre o usuário.

Na Classe ResourceServerConfig, agora é necessário substituir o login e senha que estão informados direto no código, pelo usuário e senha do banco de dados.

O código desta classe está da seguinte forma:

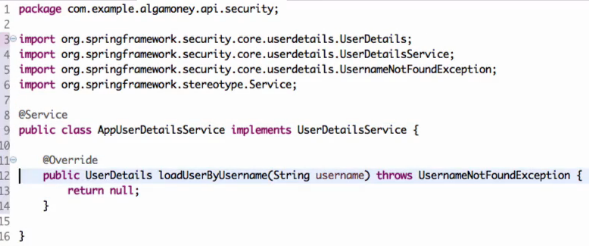


Agora deverá ser alterado para:

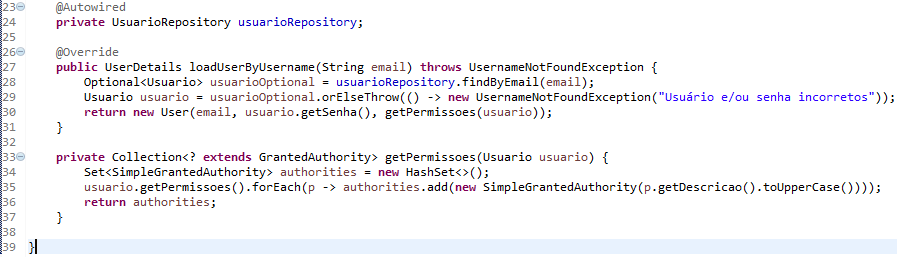


Observe que que no método configure(), a variável "auth" agora implementa o **userDetailsService**, ao invés de inMemoryAuthentication().

Agora é necessário fazer a implementação do userDetailsService. Crie a classe na pasta security, conforme abaixo:



Os métodos dessa classe ficarão da seguinte forma:

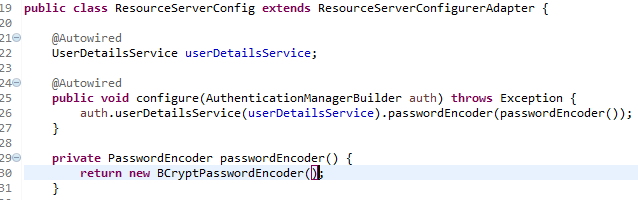


Linha 27 e 28 - Para logar no sistema o usuário deverá informar o endereço de e-mail.

Linha 29 – Se não encontrar o usuário, o Throw lança o erro para a tela e o retorno é NULL. Se usuário e senha correto, executa o return da linha 30, que é o e-mail, a senha e as permissões do usuário, as quais são 'setadas' pelo método da linha 33, conforme valores que está na tabela "Permissoes" no banco de dados.

A partir daí fica a cargo do SpringSecurity validar se o usuário e senha estão corretos. Só que para ele verificar se a senha está correta é preciso informa-lo de que modo a senha foi encriptada (encodada).

Para isso, é necessário mais uma alteração no método config() da classe ResourceServerConfig.



Na linha 26, agora temos o "passwordEncoder(passwordEncoder())", que executa o método da linha 29 e retorna um **BCryptPassordEncoder**, que foi o recurso utilizado para encriptar a senha.

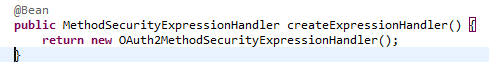
Falta agora implementar as restrições de acesso por perfil de usuário.

**32. Adicionando permissões de acesso**

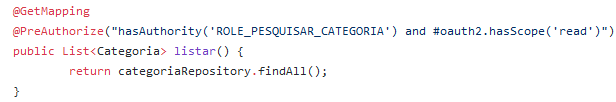
Comece adicionando a anotation "**@EnableGlobalMethodSecurity(prePostEnabled = true)**" na classe "**ResourceServerConfig**", que possibilitará implementar a segurança nos métodos das classes Categoria, Lançamentos etc. Mantenha as anotações anteriores que já estavam.



O próximo passo é adicionar a esta classe um novo **@Bean**, para que a validação utilize o recurso do OAuth2.



No método listar() da classe categoria, acrescente a **anotação que validará a permissão de acesso** do usuário que poderá listar categorias.

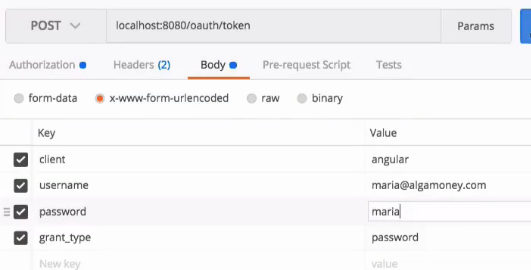


Este é o scopo de autorização da aplicação (front-end), conforme configuramos na classe **AuthorizationServerConfig**.

Note que a notação **@PreAuthorize** estabelece que somente os usuários que tenham a autorização "ROLE\_PESQUISAR\_CATEGORIA", poderão listar categorias. Esta role deverá estar com o nome idêntico ao que está cadastrado no banco de dados.

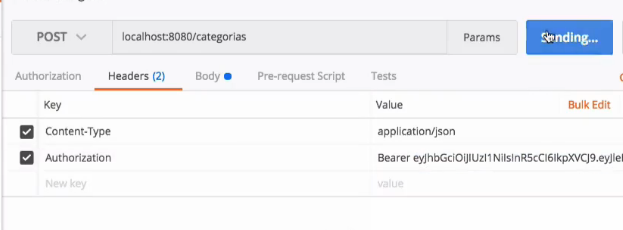
**33. Testando no POSTMSN a autorização de acesso**

Configure usuário e senha, para gerar o token.



**Figura 36** – testando autorização de acesso.

Copie e cole o novo toke, em "localhost/categorias", por exemplo e execute o teste de acesso, clicando no botão "send".



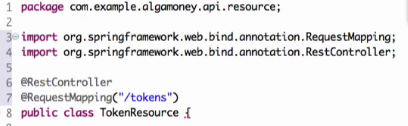
Cole aqui o novo token,

**Figura 37** – Colando o novo token gerado.

O back-end poderá ser acessado a partir de vários front-ends. Cada front-end poderá ter tipos de autorização de acesso diferentes, conforme for configurado na classe **AuthorizationServerConfig.**

**34. Implementando Logout**

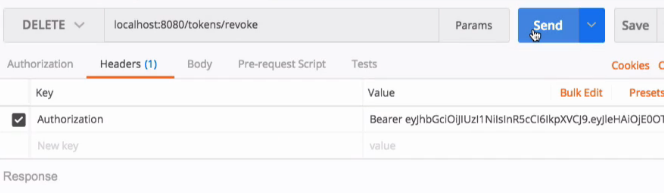
Ao clicar em "logout" o sistema deve excluir da memória o access\_token, como também excluir do cookie o refresh\_token. Para tanto, é necessário criar uma classe para gerenciar os tokens:





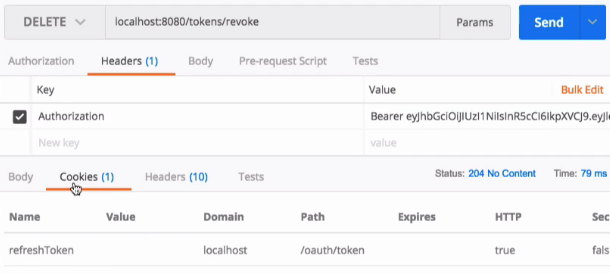
Essa classe tem um único método que exclue/invalida os tokens.

No teste a fazer com o POSTMAN, execute o localhost/token, para gerar um novo token. Copie o token gerado e cole no teste de logout, conforme abaixo. Ao clicar no botão "Send", verifique se o token foi excluído.



**Figura 39** – Teste de logout.

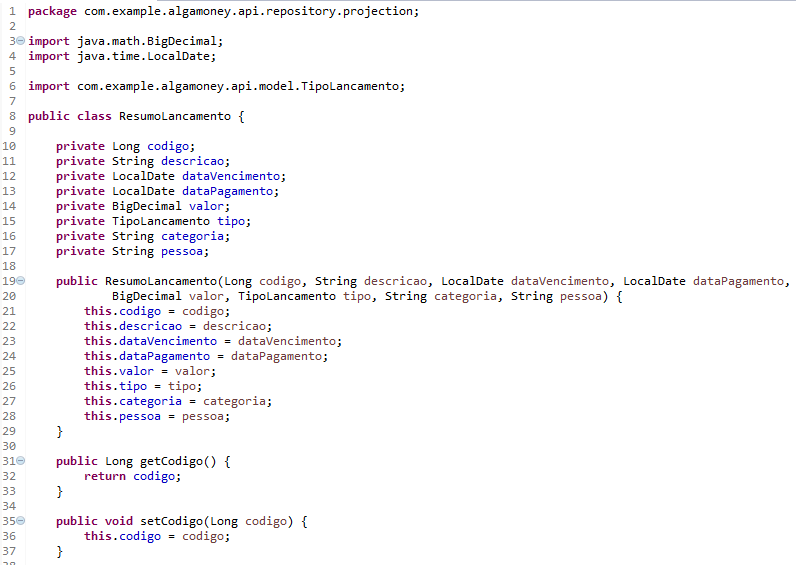
Valor do refresch\_token zerado após executar o revoke.



**Figura 40** – Refresh\_token removido.

**35. Implementando projeção de lançamento** *(aula 7.1)*

Projeção é buscar do banco de dados somente as informações que interessam para uma determinada funcionalidade. Nem sempre é necessário trazer do banco todas as informações de um registro. No caso de Lançamento poderemos criar uma projeção que seria o resumo de um lançamento.



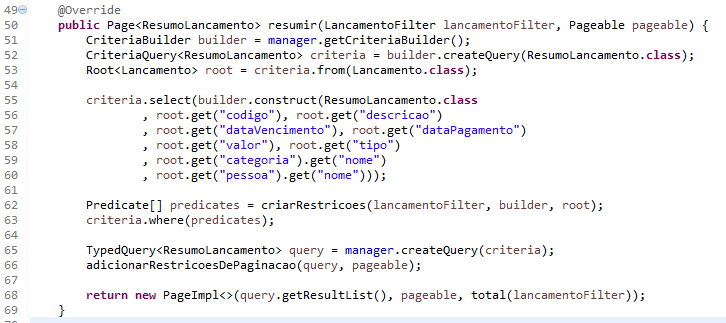
Gere os métodos GETs e SETs para todos os atributos.

Esta é considerada uma projeção (resumo) de Lançamento, pelo fato de os atributos "categoria" e "pessoa" serem do tipo "String", não está pegando também o campo "observação" e outros. Dessa maneira o sistema não vai trazer todos os registros de categoria e de pessoa, caso precisemos somente do nome da pessoa ou do nome da categoria.

Agora é necessário implementar o método que utilizará o resumo de Lançamento. Na interface "LancamentoRepositoryQuery", acrescente o método:

public Page<ResumoLancamento> resumir(LancamentoFilter lancamentoFilter, Pageable pageable);

Agora na classe LancamentoRepositoryImpl, faça a implementação do método, da seguinte forma:



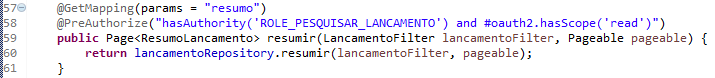
Linha 53 – define a classe de origem (from) da Pesquisa.

Linhas 56 a 60 – os atributos devem estar na mesma ordem em que estão na classe ResumoLancamento.

Foi necessário fazer uma alteração em um dos parâmetros do método "adicionarRestricoesDePaginacao()" para que ficasse mais genérico mais genérico. O nome da classe em **TypedQuery**, foi substituído por um interrogação:

private void adicionarRestricoesDePaginacao(TypedQuery<?> query, Pageable pageable) {

Agora na classe **LancamentoResource**, é necessário adicionar um método para buscar este resumo de lançamento. Para tanto, basta replicar o método de pesquisa que já existe e fazer as devidas alterações. Ao invés do nome "pesquisar", será "resumir" e "filtrar" também muda para "resumir". O retorno do método passa a ser "ResumoLancamento".

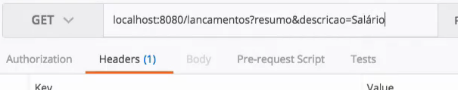


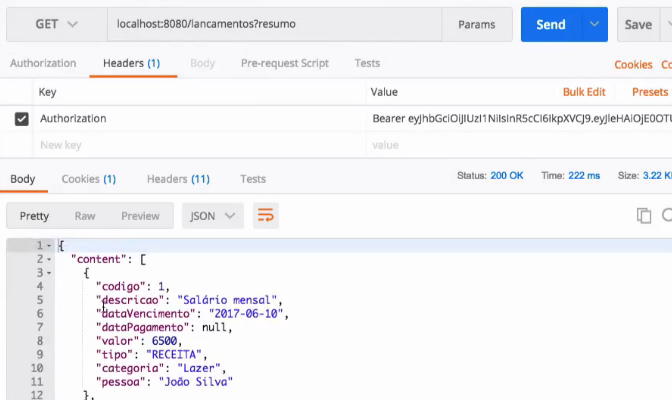
Para diferenciar o @GetMapping, entre os métodos pesquisar e resumir, foi acrescentado a este o parâmetro "resumo".

No teste com o POSTMAN basta acrescentar "**?resumo**" à url "localhost/lançamentos". Lembrando que é necessário gerar o Token, copiá-lo e cola-lo no redear desta nova pesquisa, conforme ilustra a figura 41.

Observe no resultado da pesquisa, que diferentemente da pesquisa comum, o resumo não traz todos os dados de categoria e pessoa, mas apenas o nome.

A esta pesquisa de resumoLancamento também podem ser utilizados os filtros que já implementamos. Por exemplo, listar registros que contenham o termo "Salário" na descrição:





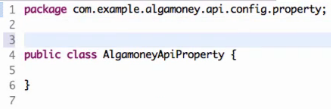
Recuperou do Banco somente o nome.

**Figura 41** – Teste de Resumo de Lançamento.

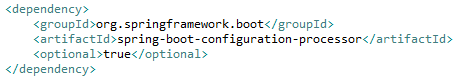
**36. Profiles do Spring** *(aula 7.2)*

Existem algumas parametrizações que fazemos no sistema, que mudam o valor de certas propriedades de acordo com o ambiente em que o sistema estiver rodando. Por exemplo, no ambiente de desenvolvimento, algumas propriedades tem valor FALSE que, em ambiente de produção, deverão assumir valor TRUE. Para evitar ter que ficar trocando estes valores manualmente, o Spring Boot tem recursos que permitem configurá-los externamente.

Primeiramente criamos a classe "AlgamoneyApiProperty dentro do pacote "property", a ser criado dentro do pacote "config".



Para implementação desta classe é necessário acrescentar uma nova dependência no Pom.xml.



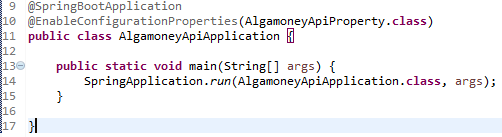
Ela está como opcional, para o caso de este projeto ser empacotado junto com outro projeto, pode ser que não seja necessária esta dependência.

À classe AlgamoneyProperty, acrescente a seguinte anotation:



Esta anotation é da dependência do Spring boot.

Na classe principal da aplicação (**AlgamoneyApiApplication**), também é necessário acrescentar uma "anotation": @EnableConfigurationProperties(AlgamoneyApiProperty.class)



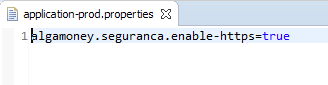
O conteúdo da classe AlgamoneyApiProperty, ficará da seguinte forma:

Este é o nome que atribuímos para esta configuração.

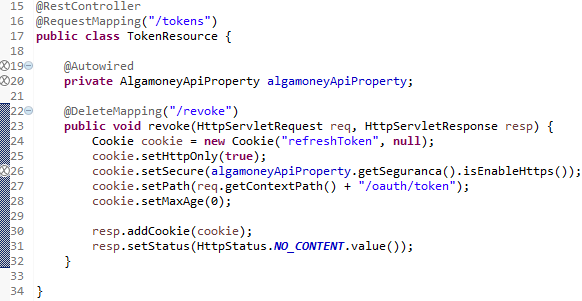


Visando melhor organização, recomenda-se separar os parâmetros de configuração por grupos (classes estáticas)

Agora já conseguimos "setar" parâmetros no arquivo application.properties. No caso, precisaremos um arquivo "properties" para o ambiente de desenvolvimento e outro para o ambiente de produção. O arquivo "application.properties" já é do ambiente de desenvolvimento, portanto precisaremos criar outro arquivo semelhante a este, para o ambiente de produção. Poderemos chama-lo de "application-prod.properties, que é a forma como o Spring o reconhece. Por enquanto este arquivo terá somente uma propriedade parametrizada, conforme abaixo:



Na classe TokenResource, agora podemos substituir o valor "False" da propriedade "Secure" pela propriedade que está no arquivo "application-prod.properties", da seguinte forma:



Observe que na linha 26, o valor da propriedade agora é "algamoneyApiProperty.getSeguranca().isEnableHttps().

Alterar também a classe " **RefreshTokenPostProcessor**". No método "**adicionarRefreshTokenNoCookie**", faça a alteração na linha do "refreshTokenCookie.setSecure", para ficar da seguinte forma:

refreshTokenCookie.setSecure(algamoneyApiProperty.getSeguranca().isEnableHttps());

Não esqueça de incluir nesta classe também o @Autowired private AlgamoneyApiProperty, conforme está na classe acima.

Na classe **CorsFilter**, também podemos parametrizar o atributo originPermitida para que tenha valor diferente de acordo com o ambiente que o sistema estiver rodando. Uma vez que já o especificamos na classe AlgamoneyApiProperty (*ver página anterior linha de código 8*), podemos fazer agora as alterações abaixo na classe CorsFilter:

Elimina-se a linha " private String originPermitida = <http://localhost:8000>.

Faz-se a injeção de dependência conforme está na linha 19 e 20 da classe TokenResource.

Altera-se o response.setHeader("Access-Control-Allow-Origin" para:

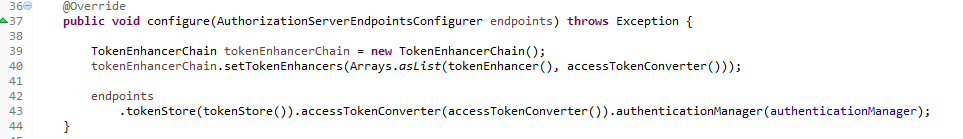
response.setHeader("Access-Control-Allow-Origin", algamoneyApiProperty.getOriginPermitida());

Altera-se o IF da classe CorsFilter para o seguinte:

if ("OPTIONS".equals(request.getMethod()) && algamoneyApiProperty.getOriginPermitida().equals(request.getHeader("Origin")))

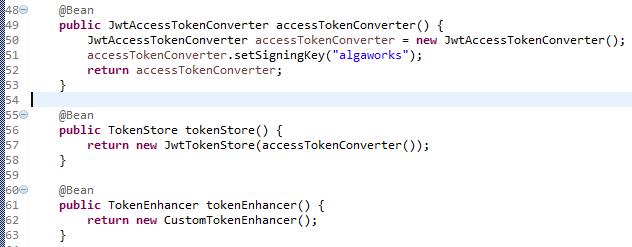
**37. Exibir o nome do usuário no topo da tela a partir do Token** *(aula 7.5)*

Na classe **AuthorizationServerConfig**, deverá ser implementado o uso de um token que armazena um número maior de elementos. Não mais será utilizado o Access\_Token.



Na linha 40 temos uma cadeia de elementos do sistema que trabalharão com o uso do Token.

Esta implementação ainda faz uso dos métodos abaixo, que também deverão estar nesta classe:



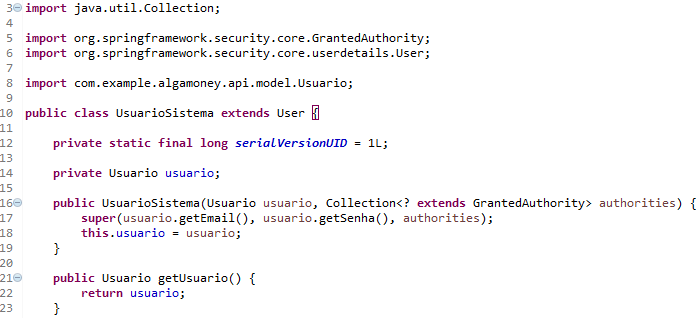
A classe CustomTokenEnhancer deverá ser implementada em "algamoney.api.config", conforme abaixo:



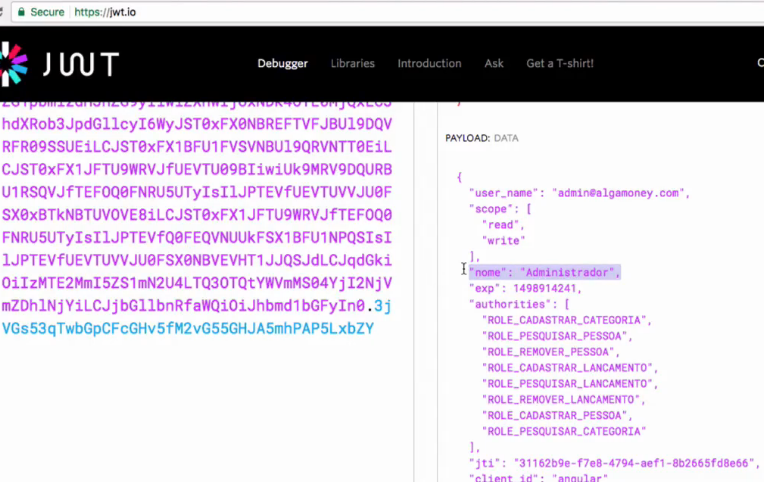
Recupera o usuário logado com **getPrincipal**()

Esta classe recupera o nome do usuário da "authentication" e armazena no token (linhas 21 e 23).

Esta classe faz uso da classe **UsuarioSistema**, que precisamos criar conforme abaixo, dentro da pasta "**algamoney.api.security**". UsuarioSistema é o usuário logado:



A partir disso, se você pegar o novo token a ser gerado no POSTMAN e visualizar seu conteúdo por meio do site <https://wt.io>, você já conseguirá ver a informação do usuário logado.



Usuário logado