**Construindo uma API REST com Java e Spring**

1. Desafio números aleatórios para loteria:
2. Como base para esse projeto vou utilizar as tecnologias:

* **Spring Boot v2.4.2:**

Para facilitar na criação e configuração inicial do projeto;

* **Spring Web:**

Com esse módulo tenho acesso a um servidor de aplicação chamado Tomcat embutido no framework, então não precisamos configurar um servidor de aplicação para executar o projeto;

* **Spring Validation:**

Disponibiliza inúmeras anotações para validação de dados de entrada da API e a criação de anotações personalizadas;

* **Spring Data JPA:**

O Spring Data usa Hibernate como provedor de implementação JPA padrão, oferece um grande suporte na implementação de camadas que acessam o banco de dados ex. (DAO), recursos como implementação de um poderoso repositório com métodos CRUD, paginação, ordenação, suporte a consultas mais complexas com JPQL e muito mais;

* **Spring test:**

fornece todo o ecossistema Spring facilitando na criação de testes de integração, mock objects,

* **Modelmapper:**

Uma biblioteca muito útil para a automatizar a conversão de classes POJO ou DTO para classes que representam entidades do banco de dados;

* **Springdoc - openapi:**

Uma biblioteca que automatiza a criação da documentação de toda a API;

* **PostgreSql:**

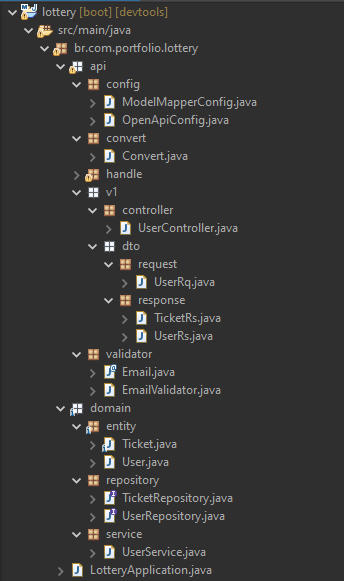
Um banco de dados relacional;

* **Docker:**

Será usado Docker e Docker-compose para encapsular tanto a aplicação como o banco de dados;

1. O projeto:

Segue abaixo a estrutura do projeto:

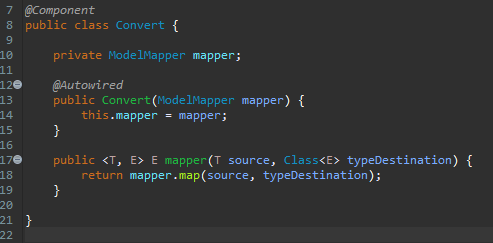


Como de costume começo pela borda mais externa da aplicação isso ajuda a ter um resultado inicial mais rapidamente, simplesmente utilizando o Postman para enviar requisições e ver se já estão chegando ao controller, então assim que crio o projeto, a primeira classe a ser criada é a UserController segue abaixo a implementação:

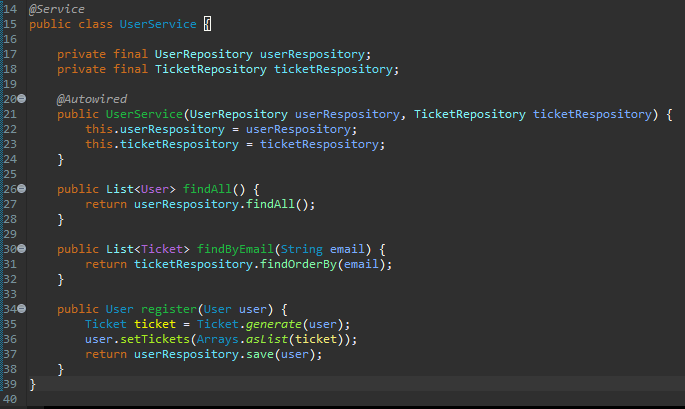


Este é o único controller da aplicação, possui 4 métodos e um construtor, o construtor recebe dois atributos por injeção de dependência, a classe da camada de serviço e uma classe com um método genérico de conversão, então podemos converter tanto classes de Entidade para DTOs como também DTOs para entidade. Merge(User user) é o único método privado dessa classe, sua função é converter a Entidade User para um DTO UserRs e todos os Tickets para TicketRs, que são objetos de resposta da aplicação;

Segue abaixo as classes recebidas pelo construtor:



Classe Convert anotada com @Component para ser gerenciada pelo Spring, possui um único método que recebe as entidades T e E que são o objeto a ser mapeado e o tipo de destino e retorna uma instancia da entidade do tipo de destino já com tudo mapeado.



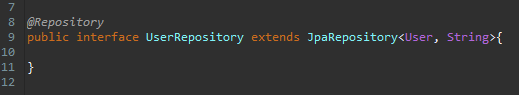
Service: a camada de serviço contém as regras de negócio da aplicação, possui 3 métodos e 1 construtor, o construtor recebe por UserRepository e TicketRepository parâmetros;

findAll() – simplesmente repassa o resultado do userRepository.findAll(), retornando todos os usuários do banco de dados;

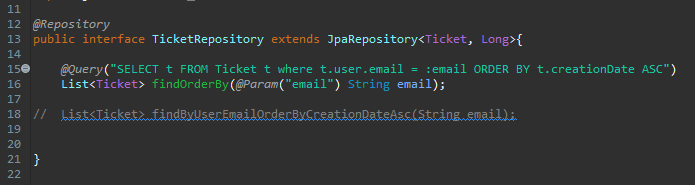
findByEmail(String email) – recebe um e-mail por parâmetro e repassa o resultado do ticketRepository.findOrderBy(email);

register(User user) – recebe um usuário por parâmetro, cria um ticket com o método Ticket.generate(user) atribuindo o ticket ao usuário, pois existe um relacionamento bidirecional , atribui também o usuário ao ticket com o user.setTicket(ticket), por fim utiliza o método userRepository.save(user) passando o usuário com todas as modificações;

Segue abaixo as classes recebidas pelo construtor:

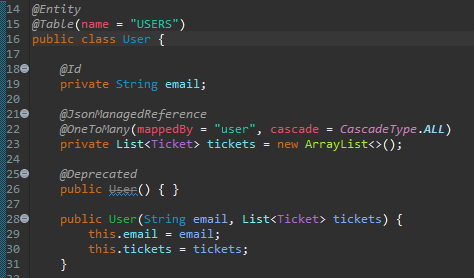


UserRepository é uma interface simples que estende JpaRepository recebendo a entidade User e o ID da User.class que no caso é uma String, com isso tenho acesso a inúmeros métodos de CRUD facilitando muito a manipulação da Entidade User no banco de dados;



TicketRepository é basicamente a mesma implementação da interface UserRepository, a única diferença é o método findByOrder que recebe um e-mail por parâmetro e possui uma anotação @Query com uma query um pouco mais complexa feita em JPQL, retorna todos os tickets do email passado por parâmetro e ordena por um atributo da classe Ticket chamado creationDate de Ascendente, outra opção também poderia ser criar um método com a nomenclatura do do SpringDataJPA, mas a desvantagem é que se eu preciso fazer uma busca um pouco mais complexa o nome do método fica enorme então optei por simplificar o nome do método e criar a query eu mesmo;

Segue abaixo as classes de entidade:



A classe User possui duas anotações:

@Entity – mapeia esta classe como uma representação da Entidade do banco de dados;

@Table – com essa anotação renomeio a tabela no banco de dados de User para USERS, pois User é uma palavra reservada do banco de dados postgreSql e é comum utilizar as palavras todas e caixa alta e no plural;

Esta classe possui dois atributos:

email – que também é o ID desta classe;

tickets – uma lista da entidade Ticket, inicio com um arrayList vazio para não ter problemas com NullPointerException. Anotada com @OneToMany dizendo que um usuário possui muitos tickets, ainda na mesma anotação digo que este atributo está mapeado na classe ticket com o nome do atributo de “user” através do parâmetro mappedBy, também digo que todas as mudanças desta classe no banco de dados podem influenciar na lista de tickets, ex. se deletarmos o usuário e ele contém tickets consequentemente será deletado os tickets, outra anotação é a @JsonManagedReference - usada para indicar que o atributo faz parte de uma ligação bidirecional, o atributo deve ter uma única propriedade compatível anotada com @JsonBackReference, isso corrige o problema de recursão infinita entre as classes que possuem esse relacionamento;

Um construtor público padrão com a anotação @deprecated para indicar que este construtor não deve ser utilizado;

Um construtor com todos os atributos da classe;

A classe também possui os métodos boilerplate (métodos padrões ex. getters, setters toString, equals e hashcode)



A classe User possui duas anotações:

@Entity

@Table

A classe Ticket possui quatro atributos:

id - Um atributo do tipo Long sequencial gerado automaticamente, com a anotação @GeneratedValue - apontando a estratégia como sequencial e aponto um gerador que será criado logo na anotação seguinte;

@SequenceGenerator - nomeio o gerador como “generator”, também nomeio a sequence no banco de dados com a sequenceName com o nome “TICKETS\_ID\_SEQ”, e indico a quantidade a ser incrementada com allocationSize=1;

User – este atributo possui 3 anotações:

@JsonBackReference - anotação citada na classe anterior, que faz com que o atributo não seja serializado, e durante a dê serialização, seu valor é definido para a instância que possui a anotação de gerenciamento

@ManyToOne – apontando que existe um relacionamento de muitos tickets para um usuário;

@JoinColumn – com o parâmetro “name” consigo nomear a chave estrangeira;

Numbers – este atributo é um conjunto de inteiros como default sempre será iniciado um HashSet vazio para evitar NullPointerException, também possui duas anotações:

@ElementCollection para trabalhar com lista no banco de dados é preciso ter essa anotação, com o parâmetro “fetch” acrescentando o valor do tipo EAGER todas as buscas feitas pelo ticket retornarão à lista em conjunto;

@CollectionTable – será criado uma tabela para gerenciar essa lista e com o parâmetro “name” consigo nomear a tabela, e com o parâmetro “joinColumns” adiciono a anotação @JoinColumn com o parâmetro “table” para nomear a chave estrangeira desta tabela;

creationDate – este atributo é do tipo “OffsetDateTime” para armazenar a hora que o ticket foi criado, tenho preferência por “OffsetDateTime” para poder disponibilizar o Time Zone assim tenho como base o Meridiano de Greenwich;

Um construtor público padrão com a anotação @deprecated para indicar que este construtor não deve ser utilizado;

Um construtor privado com todos os atributos da classe;

generate(User user) – um método do tipo “Static Factory Method” que retorna uma instância de Ticket, este método possui toda a lógica para criar um Ticket, primeiro é criado um Set de inteiros chamado “numbers”, também é criado uma instância de Random chamada “random”, crio uma estrutura de repetição dizendo para adicionar um número randômico de 1 a 60 (de acordo com a Mega-Sena) enquanto o conjunto for menor que 6, então por optar pela estrutura de dados “Set” os números não se repetem, por fim retorno o método construtor privado com o id = null, o usuário passado por parâmetro, o conjunto de números randômicos, e a data da instância do Ticket;

optei por essa estratégia pois a responsabilidade da criação de um Ticket fica na própria classe;

Static Factory Method (SFM) X Construtor:

- Com SFM consigo ter um método com um nome mais significativo;

- Com SFM consigo ter flexibilidade no retorno;

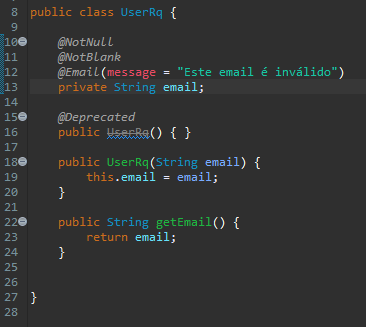
A classe também possui os métodos boilerplate (métodos padrões ex. getters, setters toString, equals e hashcode)

**Pacote DTOs:**

Com a utilização de DTOs consigo mapear as informações obtidas do banco de dados e apresentar no meu controller, ter controle do que receber e do que enviar, consigo ter controle sobre o que apresentar para não exibir dados sensíveis;

Segue abaixo as classes DTOs:

- Request:



UserRq – uma classe simples que representa um modelo de requisição da API o nome da classe é composto pelo nome da entidade correspondente e o tipo de DTO ex.:

Request = Rq;

Response = Rs;

Contém apenas um atributo:

email – um atributo do tipo String contém três anotações:

@NotNull – uma anotação do Spring Validation, valida que este campo nunca receba valores nulos;

@NotBlank – também uma anotação do Spring Validation, valida que este campo nunca receba valores em branco sem nenhum caractere;

@Email – uma anotação personalizada que valida o e-mail, recebe um parâmetro “message” enviando uma mensagem caso o e-mail não seja valido;

A classe também possui os métodos boilerplate (métodos padrões ex. getters, setters);

- Response:



UserRs – uma classe simples que representa um modelo de resposta da API;

Contém dois atributos:

email – um atributo do tipo String;

ticketsRs – um atributo do tipo lista de TicketRs (que será mostrado logo a seguir), já iniciado com ArrayList vazio para não ter problema com NullPointerException;

A classe também possui os métodos boilerplate (métodos padrões ex. getters, setters);

****

TicketRs – uma classe simples que representa um modelo de resposta da API;

Contém três atributos:

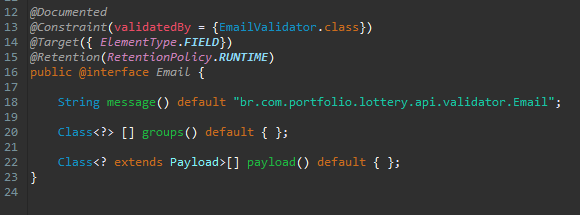
id – um atributo do tipo Long;

numbers – um atributo do tipo conjunto (Set) de inteiros, já iniciado com HashSet vazio para não ter problema com NullPointerException;

creationDate – um atributo do tipo OffsetDateTime;

A classe também possui os métodos boilerplate (métodos padrões ex. getters, setters);

**Pacote Validator:**

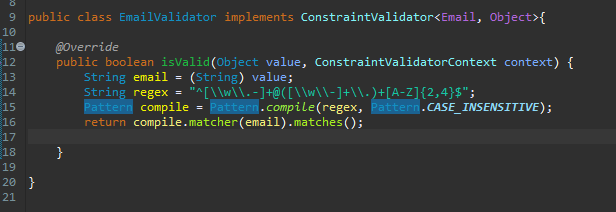
****

Anotação email:

- O alvo dessa anotação são atributos;

- É executado em tempo de execução;

- Possui uma classe de validação chamada EmailValidator;



EmailValidator – uma classe que implementa ConstraintValidator recebendo a anotação Email e um objeto;

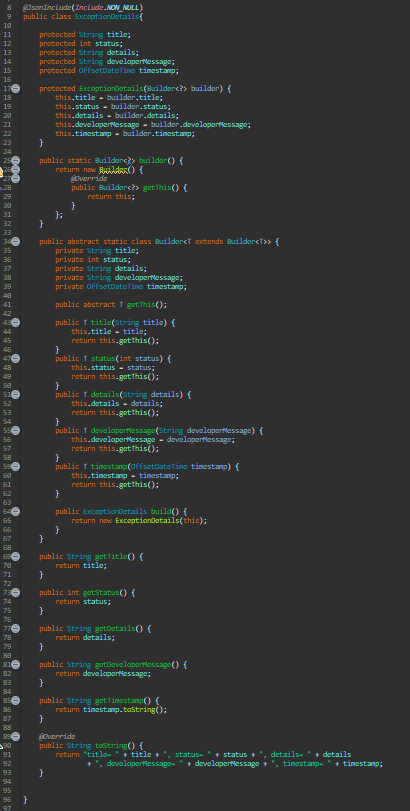
Esta classe contém um método que vem da interface;

isValid(Object value, ConstraintValidatorContext context) – este método faz um cast de Object para String, crio uma expressão regular, crio também um padrão e passo o email pelo padrão para verificar se esta validado ou não;

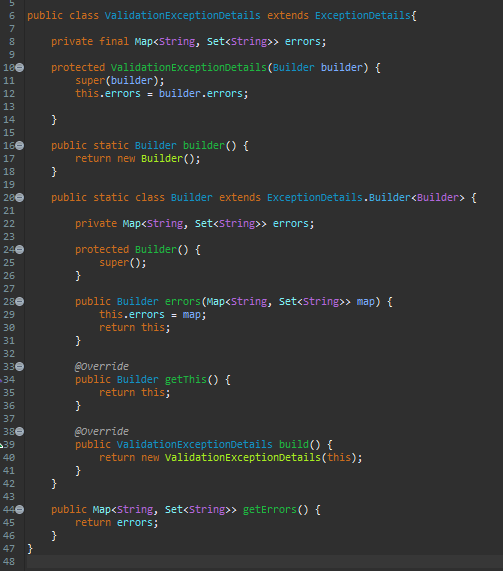
**Pacote handle:**

****

RestExceptionHandler – responsável por tratar as exceções e apresentar de forma mais amigável, extende de uma classe abstrata chamada ResponseEntityExceptionHandler;



ExceptionDetails – modelo base de apresentação de uma exceção na API;

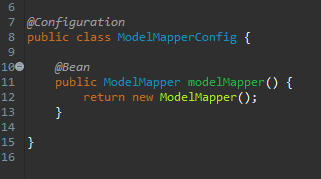


ValidationExceptionDetails – modelo mais especifico de apresentação de uma exceção na API;

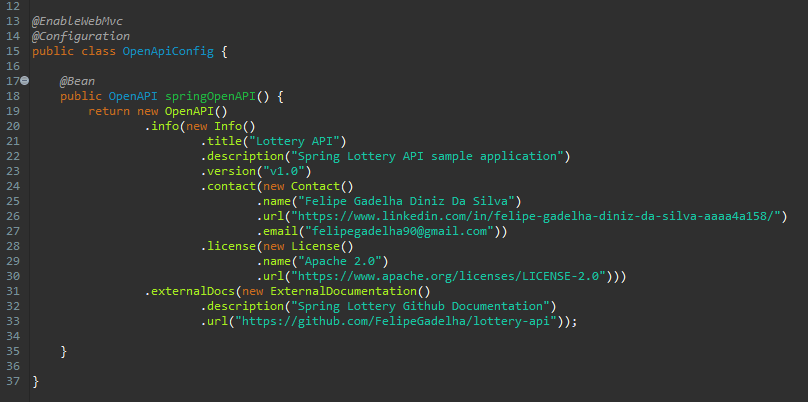


BadRequestExceptionDetails – modelo mais especifico de apresentação de uma exceção na API;

**Pacote config:**

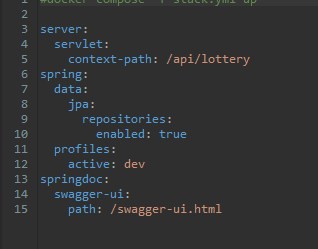


ModelMapperConfig – Uma classe de configuração que contém um Bean, para que o modelMapper seja instanciado pelo container de dependências do Spring preciso criar essa configuração;

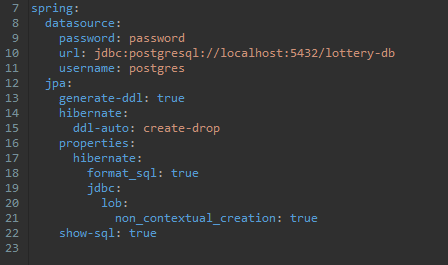


OpenApiConfig – é a classe de configuração do Springdoc-openApi;

**Configurações do Projeto:**



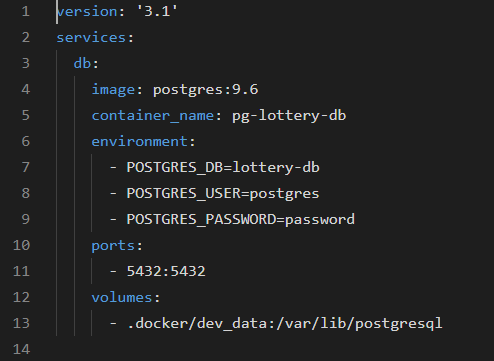
Arquivo base application.yml;



Arquivo application-dev.yml;

Com esses arquivos crio a configuração com o banco de dados;

Banco de Dados:



Docker-compose.yml – com este arquivo crio a minha instancia de um banco de dados postgreSql;

**Finalização:**

Para visualizar o projeto segue o link no github:

<https://github.com/FelipeGadelha/lottery-api>

informações de contato:

nome: Felipe Gadelha Diniz da Silva

Email – [felipegadelha90@gmail.com](mailto:felipegadelha90@gmail.com)