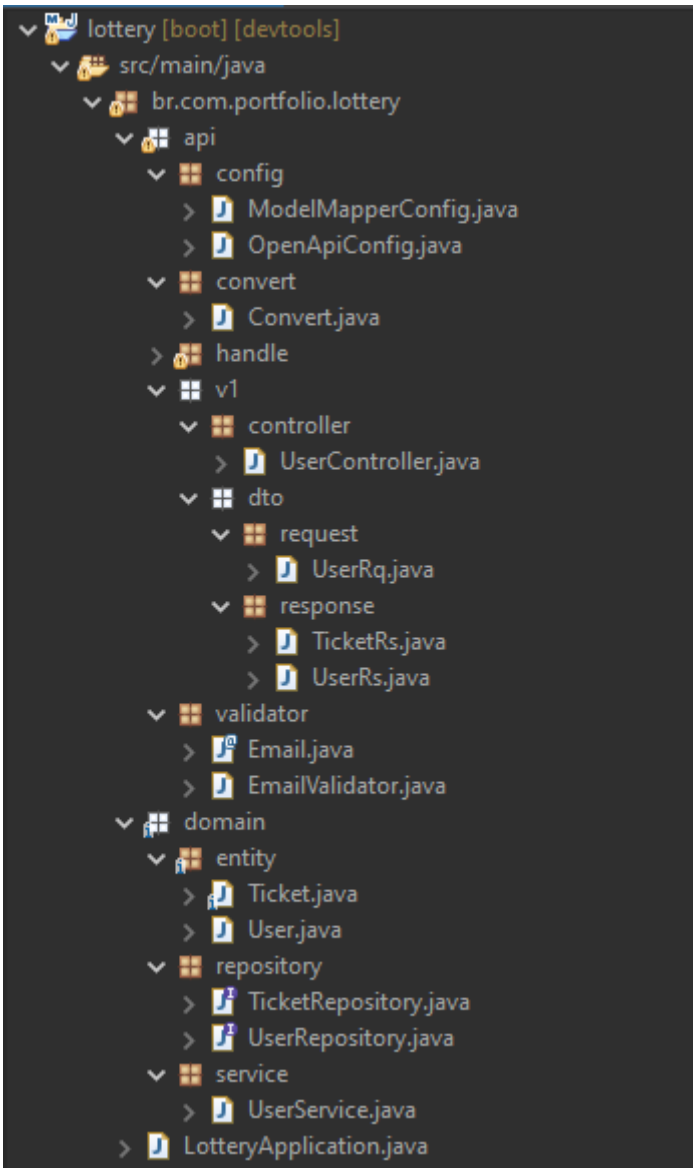
Construindo uma API REST com Java e Spring

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;

3. 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:

```java
@RestController
@RequestMapping(value = "/v1/user")
public class UserController {

    private final UserService userService;
    private final Convert convert;

    @Autowired
    public UserController(UserService userService, Convert convert) {
        this.userService = userService;
        this.convert = convert;
    }

    @GetMapping("/tickets/{email}")
    public List<TicketRs> findTicketsByEmail(@PathVariable String email) {
        return userService.findByEmail(email).stream()
                .map(t -> convert.mapper(t, TicketRs.class))
                .collect(Collectors.toList());
    }

    @GetMapping
    public List<UserRs> findAll() {
        return userService.findAll()
                .stream()
                .map(this::merge)
                .collect(Collectors.toList());
    }


    @PostMapping("/register")
    public UserRs register(@RequestBody @Valid UserRq userRq) {
        User user = convert.mapper(userRq, User.class);
        User saved = userService.register(user);
        return this.merge(saved);
    }

    private UserRs merge(User user) {
        List<TicketRs> ticketList = user.getTickets()
                .stream()
                .map(t -> convert.mapper(t, TicketRs.class))
                .collect(Collectors.toList());
        UserRs userRs = convert.mapper(user, UserRs.class);
        userRs.setTicketsRs(ticketList);
        return userRs;
    }
}
```

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:

```java
@Component
public class Convert {

    private ModelMapper mapper;

    @Autowired
    public Convert(ModelMapper mapper) {
        this.mapper = mapper;
    }

    public <T, E> E mapper(T source, Class<E> typeDestination) {
        return mapper.map(source, typeDestination);
    }

}
```

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.

```java
@Service
public class UserService {

    private final UserRepository userRespository;
    private final TicketRepository ticketRespository;

    @Autowired
    public UserService(UserRepository userRespository, TicketRepository ticketRespository) {
        this.userRespository = userRespository;
        this.ticketRespository = ticketRespository;
    }

    public List<User> findAll() {
        return userRespository.findAll();
    }

    public List<Ticket> findByEmail(String email) {
        return ticketRespository.findOrderBy(email);
    }

    public User register(User user) {
        Ticket ticket = Ticket.generate(user);
        user.setTickets(Arrays.asList(ticket));
        return userRespository.save(user);
    }
}
```

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:

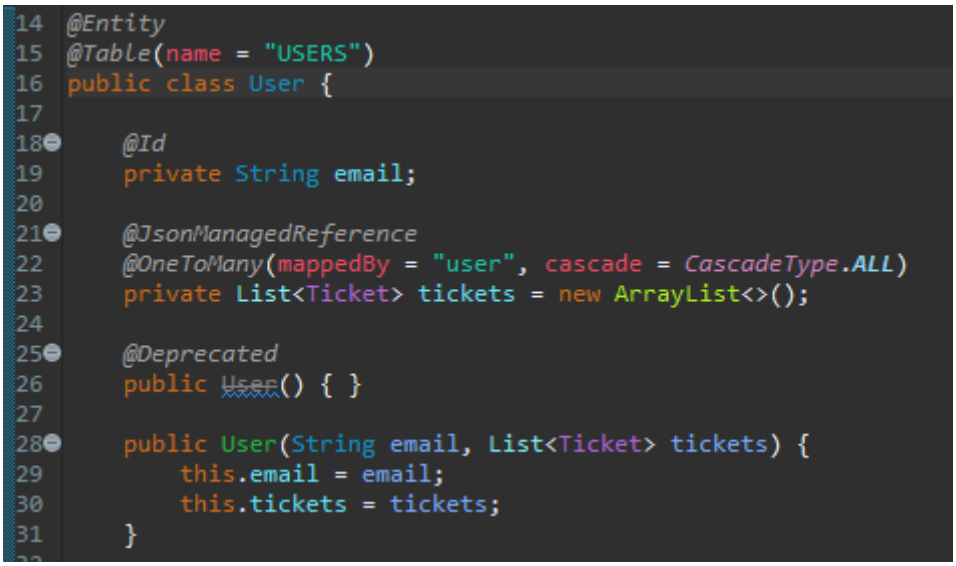
```
 7
 8  @Repository
 9  public interface UserRepository extends JpaRepository<User, String>{
10
11  }
12
```

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;

```
11
12  @Repository
13  public interface TicketRepository extends JpaRepository<Ticket, Long>{
14
15      @Query("SELECT t FROM Ticket t where t.user.email = :email ORDER BY t.creationDate ASC")
16      List<Ticket> findOrderBy(@Param("email") String email);
17
18  //  List<Ticket> findByUserEmailOrderByCreationDateAsc(String email);
19
20
21  }
22
```

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:



```java
@Entity
@Table(name = "USERS")
public class User {

    @Id
    private String email;

    @JsonManagedReference
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Ticket> tickets = new ArrayList<>();

    @Deprecated
    public User() { }

    public User(String email, List<Ticket> tickets) {
        this.email = email;
        this.tickets = tickets;
    }
```

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)

```java
@Entity
@Table(name = "TICKETS")
public class Ticket {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="generator")
    @SequenceGenerator(name="generator", sequenceName="TICKETS_ID_SEQ", allocationSize=1)
    @Column(name="id")
    private Long id;

    @JsonBackReference
    @ManyToOne
    @JoinColumn(name = "user_email")
    private User user;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "tickets_numbers",
            joinColumns = @JoinColumn(table = "ticket_id" ))
    private Set<Integer> numbers = new HashSet<>();
    @Column(name="creation_date")
    private OffsetDateTime creationDate;

    @Deprecated
    public Ticket() {   }

    private Ticket(Long id, User user, Set<Integer> numbers, OffsetDateTime creationDate) {
        this.id = id;
        this.user = user;
        this.numbers = numbers;
        this.creationDate = creationDate;
    }

    public static Ticket generate(User user) {
        Set<Integer> numbers = new HashSet<>(6);
        Random random = new Random();
        while (numbers.size() < 6) {
            numbers.add(random.nextInt(60) + 1);
        }
        return new Ticket(null, user, numbers, OffsetDateTime.now());
    }

```

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:

```java
public class UserRq {

    @NotNull
    @NotBlank
    @Email(message = "Este email é inválido")
    private String email;

    @Deprecated
    public UserRq() { }

    public UserRq(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }


}

```

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:

```java
public class UserRs {

    private String email;
    private List<TicketRs> ticketsRs = new ArrayList<>();

    @Deprecated
    public UserRs() {}

    public UserRs(String email, List<TicketRs> ticketRs) {
        this.email = email;
        this.ticketsRs = ticketRs;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public List<TicketRs> getTicketsRs() {
        return ticketsRs;
    }

    public void setTicketsRs(List<TicketRs> ticketsRs) {
        this.ticketsRs = ticketsRs;
    }

}
```

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);

```java
public class TicketRs {

    private Long id;

    private Set<Integer> numbers = new HashSet<>();

    private OffsetDateTime creationDate;

    @Deprecated
    public TicketRs() { }

    public TicketRs(Long id, Set<Integer> numbers, OffsetDateTime creationDate) {
        this.id = id;
        this.numbers = numbers;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Set<Integer> getNumbers() {
        return numbers;
    }
    public void setNumbers(Set<Integer> numbers) {
        this.numbers = numbers;
    }

    public String getCreationDate() {
        if (Objects.isNull(creationDate)) return null;
        return creationDate.toString();
    }

    public void setCreationDate(OffsetDateTime creationDate) {
        this.creationDate = creationDate;
    }

}
```

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:

```java
@Documented
@Constraint(validatedBy = {EmailValidator.class})
@Target({ ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Email {

    String message() default "br.com.portfolio.lottery.api.validator.Email";

    Class<?> [] groups() default { };

    Class<? extends Payload>[] payload() default { };
}

```

Anotação email:

- O alvo dessa anotação são atributos;
- É executado em tempo de execução;
- Possui uma classe de validação chamada EmailValidator;

```java
public class EmailValidator implements ConstraintValidator<Email, Object>{

    @Override
    public boolean isValid(Object value, ConstraintValidatorContext context) {
        String email = (String) value;
        String regex = "^[\\w\\.-]+@([\\w\\-]+\\.)+[A-Z]{2,4}$";
        Pattern compile = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
        return compile.matcher(email).matches();

    }

}

```

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;

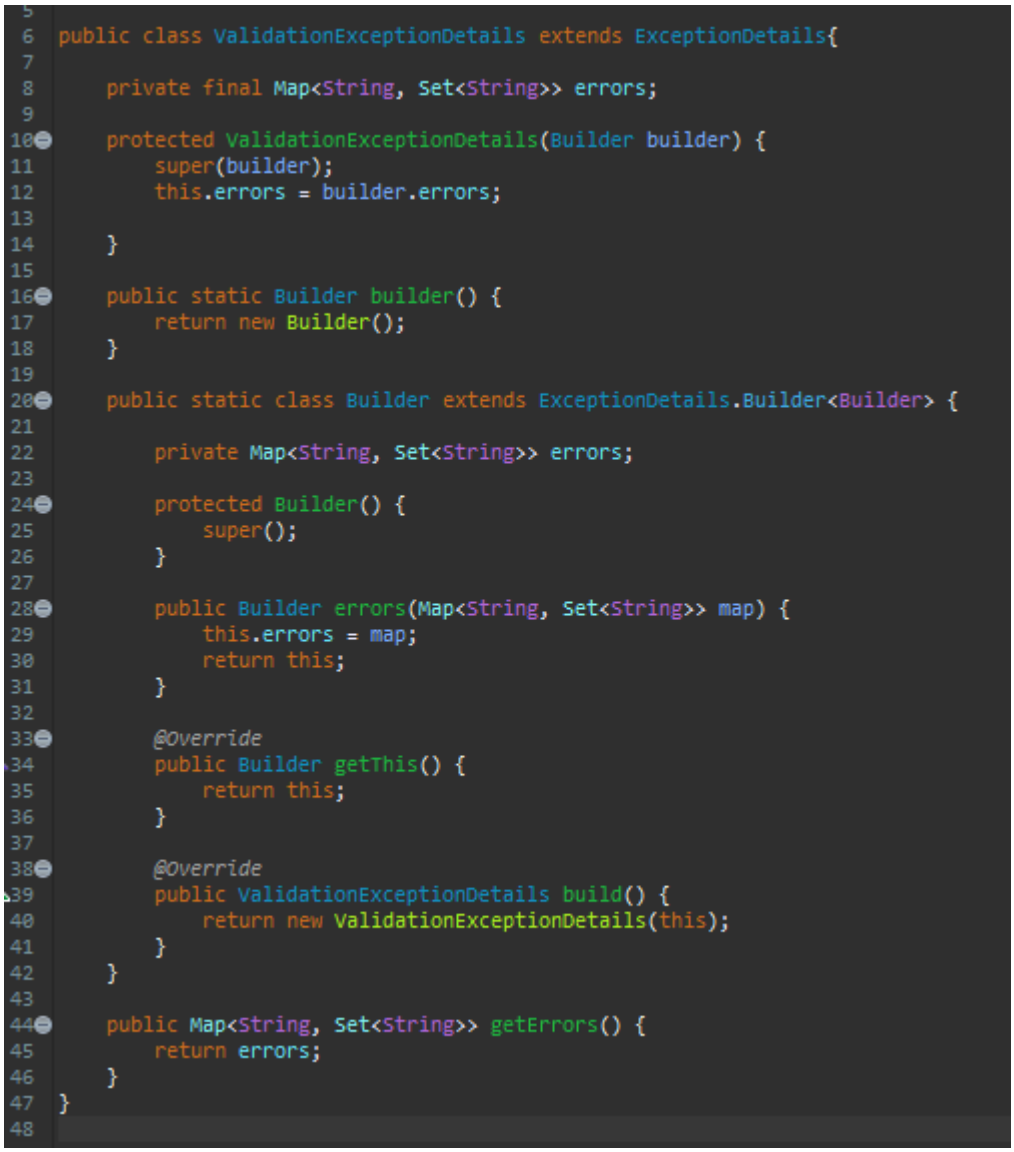
Pacote handle:

```java
@ControllerAdvice
public class RestExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(NullPointerException.class)
    public ResponseEntity<BadRequestExceptionDetails> handleNullPointerException(
            NullPointerException exception) {

        return new ResponseEntity<>(
                BadRequestExceptionDetails.builder()
                    .timestamp(OffsetDateTime.now())
                    .status(HttpStatus.BAD_REQUEST.value())
                    .title("Bad Request Exception, Check the Documentation")
                    .details(exception.getMessage())
                    .developerMessage(exception.getClass().getName())
                    .build(),
                HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<BadRequestExceptionDetails> handleIllegalArgumentException(
            IllegalArgumentException exception) {

        return new ResponseEntity<>(
                BadRequestExceptionDetails.builder()
                    .timestamp(OffsetDateTime.now())
                    .status(HttpStatus.BAD_REQUEST.value())
                    .title("Bad Request Exception, Check the Documentation")
                    .details(exception.getMessage())
                    .developerMessage(exception.getClass().getName())
                    .build(),
                HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(StaleObjectStateException.class)
    public ResponseEntity<BadRequestExceptionDetails> handleStaleObjectStateException(
            StaleObjectStateException exception) {

        return new ResponseEntity<>(
                BadRequestExceptionDetails.builder()
                    .timestamp(OffsetDateTime.now())
                    .status(HttpStatus.BAD_REQUEST.value())
                    .title("Bad Request Exception, Check the Documentation")
                    .details(exception.getMessage())
                    .developerMessage(exception.getClass().getName())
                    .build(),
                HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(IllegalStateException.class)
    public ResponseEntity<BadRequestExceptionDetails> handleIllegalStateException(IllegalStateException exception) {

        return new ResponseEntity<>(
                BadRequestExceptionDetails.builder()
                .timestamp(OffsetDateTime.now())
                .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
                .title("Bad Request Exception, Check the Documentation")
                .details(exception.getMessage())
                .developerMessage(exception.getClass().getName())
                .build(), HttpStatus.INTERNAL_SERVER_ERROR);
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException exception,
            HttpHeaders headers, HttpStatus status, WebRequest request) {

        List<FieldError> fieldErros = exception.getBindingResult().getFieldErrors();

        List<ObjectError> globalErrors = exception.getBindingResult().getGlobalErrors();

        Map<String, Set<String>> map = fieldErros.stream().collect(Collectors.groupingBy(FieldError::getField,
                Collectors.mapping(FieldError::getDefaultMessage, Collectors.toSet())));

        if (map.isEmpty()) {
            map = globalErrors.stream().collect(Collectors.groupingBy(ObjectError::getCode,
                    Collectors.mapping(ObjectError::getDefaultMessage, Collectors.toSet())));
        }
        return new ResponseEntity<>(ValidationExceptionDetails
                .builder().timestamp(OffsetDateTime.now())
                .status(HttpStatus.BAD_REQUEST.value())
                .title("Bad Request Exception, Check the Documentation")
                .details("Check the error field(s)")
                .developerMessage(exception.getClass().getName())
                .errors(map)
                .build(), headers, HttpStatus.BAD_REQUEST);
    }

    @Override
    protected ResponseEntity<Object> handleExceptionInternal(Exception ex, @Nullable Object body, HttpHeaders headers,
            HttpStatus status, WebRequest request) {
        boolean isEmpty = false;
//      InvalidDataAccessApiUsageException
        return new ResponseEntity<>(ExceptionDetails
                .builder()
                .timestamp(OffsetDateTime.now())
                .status(status.value())
                .title(isEmpty ? ex.getCause().getMessage() : ex.getMessage())
                .details(ex.getMessage())
                .developerMessage(ex.getClass().getName())
                .build(), headers, status);
    }
}
```

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

```java
@JsonInclude(Include.NON_NULL)
public class ExceptionDetails{

    protected String title;
    protected int status;
    protected String details;
    protected String developerMessage;
    protected OffsetDateTime timestamp;

    protected ExceptionDetails(Builder<?> builder) {
        this.title = builder.title;
        this.status = builder.status;
        this.details = builder.details;
        this.developerMessage = builder.developerMessage;
        this.timestamp = builder.timestamp;
    }

    public static Builder<?> builder() {
        return new Builder() {
            @Override
            public Builder<?> getThis() {
                return this;
            }
        };
    }

    public abstract static class Builder<T extends Builder<T>> {
        private String title;
        private int status;
        private String details;
        private String developerMessage;
        private OffsetDateTime timestamp;

        public abstract T getThis();

        public T title(String title) {
            this.title = title;
            return this.getThis();
        }
        public T status(int status) {
            this.status = status;
            return this.getThis();
        }
        public T details(String details) {
            this.details = details;
            return this.getThis();
        }
        public T developerMessage(String developerMessage) {
            this.developerMessage = developerMessage;
            return this.getThis();
        }
        public T timestamp(OffsetDateTime timestamp) {
            this.timestamp = timestamp;
            return this.getThis();
        }

        public ExceptionDetails build() {
            return new ExceptionDetails(this);
        }
    }

    public String getTitle() {
        return title;
    }

    public int getStatus() {
        return status;
    }

    public String getDetails() {
        return details;
    }

    public String getDeveloperMessage() {
        return developerMessage;
    }

    public String getTimestamp() {
        return timestamp.toString();
    }

    @Override
    public String toString() {
        return "title= " + title + ", status= " + status + ", details= " + details
                + ", developerMessage= " + developerMessage + ", timestamp= " + timestamp;
    }


}
```

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



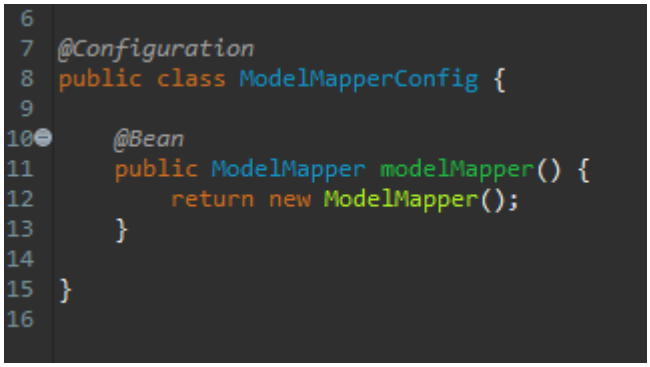
```java
public class ValidationExceptionDetails extends ExceptionDetails{

    private final Map<String, Set<String>> errors;

    protected ValidationExceptionDetails(Builder builder) {
        super(builder);
        this.errors = builder.errors;

    }

    public static Builder builder() {
        return new Builder();
    }

    public static class Builder extends ExceptionDetails.Builder<Builder> {

        private Map<String, Set<String>> errors;

        protected Builder() {
            super();
        }

        public Builder errors(Map<String, Set<String>> map) {
            this.errors = map;
            return this;
        }

        @Override
        public Builder getThis() {
            return this;
        }

        @Override
        public ValidationExceptionDetails build() {
            return new ValidationExceptionDetails(this);
        }
    }

    public Map<String, Set<String>> getErrors() {
        return errors;
    }
}
```

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

```java
public class BadRequestExceptionDetails extends ExceptionDetails{


    public BadRequestExceptionDetails(Builder builder) {
        super(builder);
    }

    public static Builder builder() {
        return new Builder();
    }

    public static class Builder extends ExceptionDetails.Builder<Builder> {

        protected Builder() {
            super();
        }

        @Override
        public Builder getThis() {
            return this;
        }

        @Override
        public BadRequestExceptionDetails build() {
            return new BadRequestExceptionDetails(this);
        }
    }

    @Override
    public String toString() {
        return super.toString();
    }


}

```

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

Pacote config:



```java

@Configuration
public class ModelMapperConfig {

    @Bean
    public ModelMapper modelMapper() {
        return new ModelMapper();
    }

}

```

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;

```java

@EnableWebMvc
@Configuration
public class OpenApiConfig {

    @Bean
    public OpenAPI springOpenAPI() {
        return new OpenAPI()
                .info(new Info()
                        .title("Lottery API")
                        .description("Spring Lottery API sample application")
                        .version("v1.0")
                        .contact(new Contact()
                                .name("Felipe Gadelha Diniz Da Silva")
                                .url("https://www.linkedin.com/in/felipe-gadelha-diniz-da-silva-aaaa4a158/")
                                .email("felipegadelha90@gmail.com"))
                        .license(new License()
                                .name("Apache 2.0")
                                .url("https://www.apache.org/licenses/LICENSE-2.0")))
                .externalDocs(new ExternalDocumentation()
                        .description("Spring Lottery Github Documentation")
                        .url("https://github.com/FelipeGadelha/lottery-api"));

    }

}
```

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

Configurações do Projeto:

```yaml
#docker-compose -f stack.yml up

server:
  servlet:
    context-path: /api/lottery
spring:
  data:
    jpa:
      repositories:
        enabled: true
  profiles:
    active: dev
springdoc:
  swagger-ui:
    path: /swagger-ui.html
```

Arquivo base application.yml;

```yaml
spring:
  datasource:
    password: password
    url: jdbc:postgresql://localhost:5432/lottery-db
    username: postgres
  jpa:
    generate-ddl: true
    hibernate:
      ddl-auto: create-drop
    properties:
      hibernate:
        format_sql: true
        jdbc:
          lob:
            non_contextual_creation: true
    show-sql: true

```

Arquivo application-dev.yml;

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

Banco de Dados:

```yaml
version: '3.1'
services:
  db:
    image: postgres:9.6
    container_name: pg-lottery-db
    environment:
      - POSTGRES_DB=lottery-db
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
    ports:
      - 5432:5432
    volumes:
      - .docker/dev_data:/var/lib/postgresql

```

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