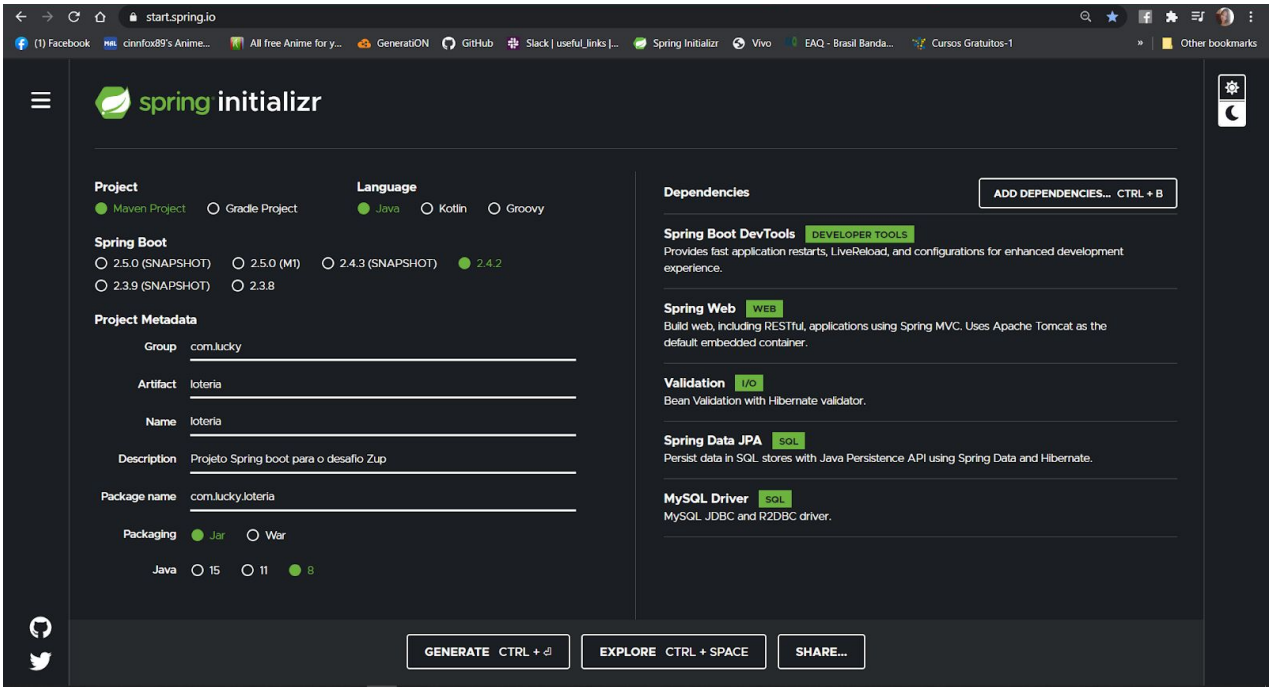
Orange Talents Zup por Cinthia Tengan

Para este desafio, foi proposto a criação de uma API REST para geração de números aleatórios de um sistema de loteria que devem ser anexados a um email para facilitar na identificação do apostador. Como requisitos foram propostos: construir um endpoint para receber o e-mail da pessoa e retornar um objeto de resposta com os números sorteados para a aposta. As informações devem ser gravadas em um banco de dados e devidamente associadas à pessoa. Também deve ser construído um segundo endpoint para listar todas as apostas de um solicitante, passando seu e-mail como parâmetro, o sistema deverá retornar em ordem de criação todas as suas apostas.

Iniciando o projeto

Para iniciar o projeto, foi utilizado a IDE eclipse, com Java versão 15.0.1, mySQL para o armazenamento em banco de dados e por fim o Postman para testar as requisições dos endpoints. Começamos utilizando o site start.spring.io que oferece algumas configurações pré estabelecidas para o projeto, como: tipo de projeto, tipo de linguagem, etc. Grande parte das configurações utilizadas foram pré-estabelecidas pelo próprio site, apenas a escolha da versão 8 do Java foi priorizada para a funcionalidade do projeto em outras máquinas que não utilizam a versão 15 do Java. Antes de gerar o projeto spring no site, adicionamos as dependências do spring que necessitamos para que ocorra a conexão do spring com o MySQL, além do JPA Hibernate. Neste projeto foram utilizados o Spring Boot DevTools, Spring Web, Spring Data JPA, Validation e MySQL driver.



Após gerar o arquivo no site, é necessário descompactar o arquivo e importá-lo no eclipse, através de *import existing project maven* (importar projeto existente de maven). Com os arquivos importados, o primeiro passo é alterar o arquivo *application.properties* (propriedades de aplicação) para que o JPA Hibernate possa enviar os dados ao banco de dados no MySQL.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost/loteria?createDatabaseIfNotExist=true&serverTimezone=UTC&useSSl=false
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.show-sql=true
```

A partir destas configurações, asseguramos que o spring fará uma conexão com o MySQL através do *localhost* (servidor local) e criará o banco de dados “loteria”, caso este não exista ainda. As configurações de usuário e senha são as mesmas utilizadas pelo MySQL da máquina local.

Criação da model Aposta

Após configurar todas as medidas iniciais para garantir boa inicialização do projeto, começamos a confecção da camada modelo com a classe Aposta. Para que a camada seja reconhecida como um modelo, utilizamos a anotação *@Entity* no início do código para que a camada seja mapeada como uma entidade no banco de dados. Em seguida, *@Table* foi utilizado para nomear a tabela que será criada no banco de dados do MySQL com os atributos da camada modelo. Então foram criados os atributos de acordo com os requisitos do projeto, como id, nome, email e número(aleatório) cada um com seu tipo e algumas validações. Já as anotações *@Id/@GeneratedValue* significam que o atributo id será a primary key da tabela e será gerado automaticamente usando a estratégia IDENTITY. Também utilizamos as anotações para validação do Bean Validation: *@NotNull* para não permitir que o valor do atributo seja nulo e *@Email* no atributo email, para validar um atributo email que contenha @. Por fim, criamos os *getters* e *setters* da nossa camada modelo para retornar os valores dos atributos da classe.

```java
package com.lucky.loteria.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotNull;

@Entity
@Table(name = "tb_aposta")
public class Aposta {

    // atributos
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @NotNull
    private String nome;
    @NotNull
    @Email
    private String email;
    @NotNull
    private long numero;

    // getters & setters
    public long getId() {
        return id;
    }

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

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

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

    public long getNumero() {
        return numero;
    }

    public void setNumero(long numero) {
        this.numero = numero;
    }

}
```

Criação da camada ApostaRepository

Na nossa aplicação necessitamos de uma camada de repositório, cujo objetivo é o armazenamento encapsulado dos parâmetros que desejamos alterar ou simplesmente guardar. Para criar o repositório, criamos uma interface com o nome ApostaRepository, que irá conter as regras de negócios do projeto. Começamos utilizando *extends* para herdar o *JpaRepository,* uma outra interface que provê a ligação a determinada classe do modelo com possibilidade de persistir no banco de dados. Utilizamos o *@Repository* para indicar ao spring que esta interface é um repositório. E como herdamos a interface do Jpa Hibernate, podemos utilizar alguns contratos já existentes dentro da JpaRepository. Portanto através de *query methods* podemos realizar consultas padronizadas em cima do banco de dados, e a utilizada no código foi o *findByEmail* (busca por email), por ser um dos requisitos de endpoints propostos no desafio.

```java
package com.lucky.loteria.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.lucky.loteria.model.Aposta;

@Repository
public interface ApostaRepository extends JpaRepository<Aposta, Long> {
    public Optional<Aposta> findByEmail(String aposta);

}

```

Criação da camada ApostaService

Para a implementação da lógica no modelo de Spring MVC, foi criada uma camada de serviço (service), separada do Rest Controller e chamada ApostaService para este projeto. Nossa primeira anotação é a *@Service* para manter a lógica de negócios da classe criada (serviço), em seguida utilizamos o *@Autowired* para fazer a injeção do repositório ApostaRepository para salvar as alterações feitas pelos métodos da classe. A lógica implementada para gerar o número aleatório para loteria foi criar um método chamado *ApostaRand,* que recebe aposta como parâmetro. Dentro do método, através de um vetor de 6 casas são preenchidos 6 valores aleatórios multiplicados por 100 cada um e armazenados em cada casa. A multiplicação por 100 ao valor aleatório foi implementada devido ao fato da função *math.random* trabalhar com números do tipo double entre 0.0 a 1.0 e então temos como retorno do método, os valores aleatórios salvos dentro do repositório.

```java
package com.lucky.loteria.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class ApostaService {

    @Autowired
    private ApostaRepository repository;

    // gerando valores aleatorios para aposta
    public Aposta ApostaRand(Aposta aposta) {
        double vetornum[] = new double[6];
        for(int i=0; i<6; i++) {
            vetornum[i] = Math.random()*100;
        }
        return repository.save(aposta); //salva o objeto apostador com o numero modificado
    }

}

```

Criação da camada ApostaController

Um Controller é responsável tanto por receber requisições como por enviar a resposta ao usuário através de protocolos http, neste projeto foi criada a classe ApostaController. A primeira notação utilizada é a *@RestController* que informa ao spring que a classe é um controller e que também contém outra anotação de *@ResponseBody* incluída. Já a segunda anotação *@RequestMapping* define a URI pela qual essa classe é acessada e *@CrossOrigin* serve para que independente de qual origem que venha a requisição, a API aceite essa requisição. A anotação *@Autowired* injeta os repositórios de ApostaRepository e ApostaService dentro do controller. Para o primeiro método de Post, que irá cadastrar a aposta do usuário, utilizamos a notação *@PostMapping* que mapeia o modo *Post* http, com os parâmetros de *ResponseEntity* (uma entidade de resposta completa) que retorna uma lista do tipo aposta. A requisição do valor desejado pelo usuário é feita pela anotação *@RequestBody* para obter o valor do corpo da requisição. No retorno do método, recebemos novamente o objeto *ResponseEntity* e o status Http de criado (201), também é retornado os dados salvos dentro do repositório junto com um número aleatório anexado à requisição. Ou seja, quando cadastramos um usuário com email e nome, ao enviar ao banco de dados, o método criado na camada de serviço já anexa a este cadastro uma aposta com 6 números. Já no método get, temos a anotação *@GetMapping* que serve para mapear o modo *Get* http, com parâmetros de retorno das informações dos valores de dentro do repositório. Este método também utiliza a anotação de *@PathVariable* na requisição de parâmetro para realizar requisições através da URI informada. Já o último método criado no controller é outro método get, com a diferença de que este retorna apenas as apostas realizadas por email com a requisição de URI.

```java
package com.lucky.loteria.controller;

import java.util.List;

@RestController
@RequestMapping("/loteria")
@CrossOrigin(origins = "*", allowedHeaders = "*")
public class ApostaController {

    @Autowired  //injeção das interfaces
    private ApostaService apostadorService;
    @Autowired  //injeção das interfaces
    private ApostaRepository repository;

    //Método post para postar uma aposta nova
    @PostMapping
    public ResponseEntity<Aposta> postApostas(@RequestBody Aposta aposta){
        return ResponseEntity.status(HttpStatus.CREATED).body(apostadorService.ApostaRand(aposta));
    }
    //Método get para trazer todas as apostas feitas
    @GetMapping
    public ResponseEntity<List<Aposta>> findAllApostas(){
        return ResponseEntity.ok(repository.findAll());
    }
    //Método get para trazer as apostas feitas pelo email
    @GetMapping("/{email}")
    public ResponseEntity<Aposta> findByApostas(@PathVariable long email){
        return repository.findById(email).map(resp -> ResponseEntity.ok(resp)).orElse(ResponseEntity.notFound().build());
    }

}

```

Implementação do sistema na Web

Antes de implementar o sistema na Web seria necessário a implementação de camadas de segurança com o spring security, para que fosse criado um sistema de cadastro com login e senha para cada usuário. Logo, seria possível criar um sistema de login com token para cada usuário e proteção com encriptação de senhas por meio de hash code. Essa medida de segurança é primordial para assegurar que os dados do usuário estejam protegidos de terceiros e para que a aplicação seja segura antes de poder ser implementada na web. Já a implementação pode ser realizada através de empacotamento e contêineres com o docker, pois por se tratar de um ambiente isolado e portátil, o deploy se torna simplificado na sua implementação.

Considerações finais

Como iniciante na área de desenvolvimento**,** minha opinião pessoal sobre o desafio proposto é que o desafio foi justo, mas de forma abrangente, sendo possível chegar a várias soluções de um mesmo problema. Portanto, nesta resolução a abordagem utilizada foi a adquirida durante o bootcamp da Generation Brasil, que acredito ser uma abordagem mais simples, porém mais assertiva para melhor desempenho de código. Particularmente foi uma experiência muito enriquecedora criar o crud e explicar passo a passo e como funciona cada camada criada, pude aprimorar meus conhecimentos sobre o próprio Spring durante o desafio.