Construindo uma API RESTful com Java e Spring Framework

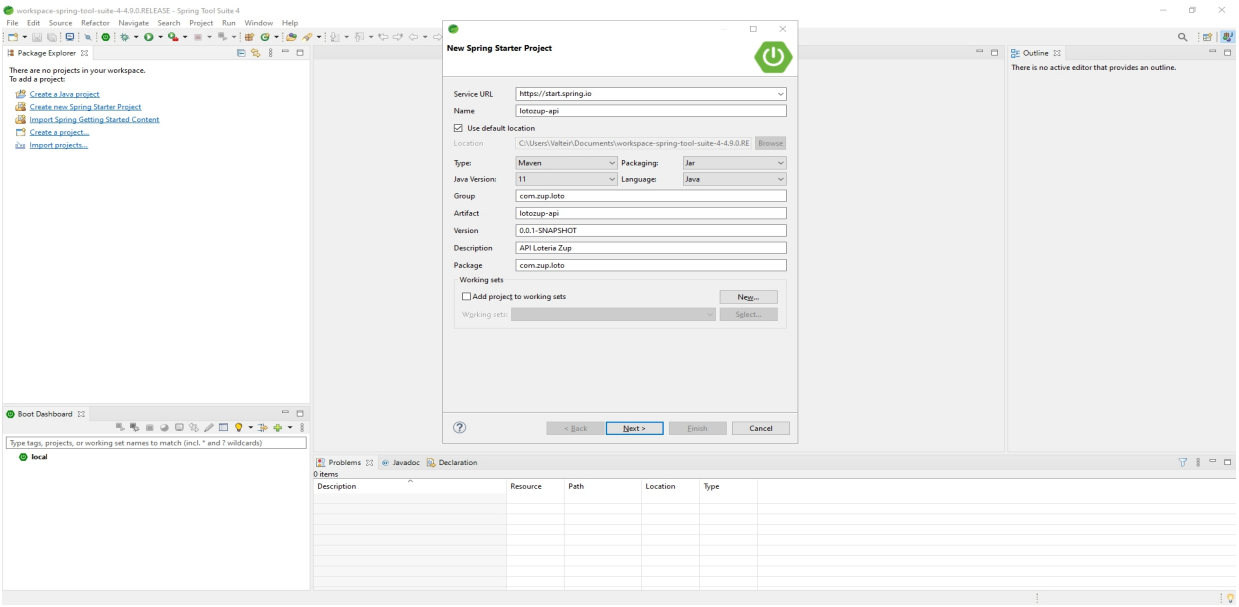
Vamos desenvolver um API que precisará gerar números aleatórios para loteria. Os números serão vinculados a uma aposta e cada aposta será vinculada a um email.
Haverá um endpoint que receberá um email da pessoa e retornará um objeto de resposta com os números sorteados. Também haverá um endpoint onde poderá ser consultadas todas as apostas de um solicitante, passando seu email como parâmetro.

Começando

Como IDE vamos utilizar a Spring Tools Suite, ela possui diversas ferramentas do ecossistema spring.

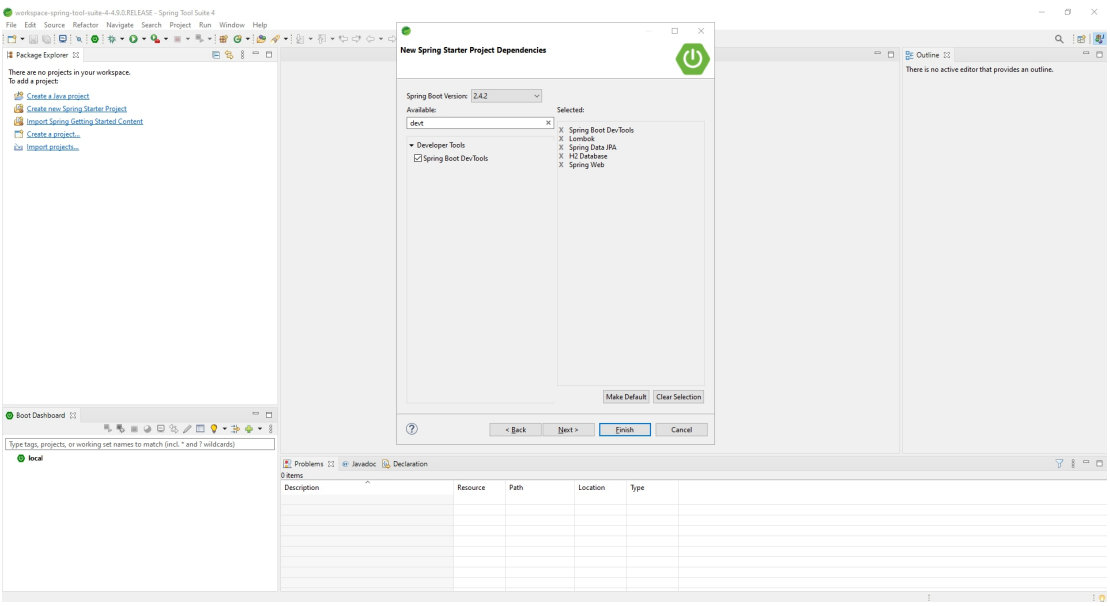
No package explorer clicamos com o botão direito > new > Spring Starter Project, nessa tela podemos definir o nome do projeto, qual linguagem utilizaremos, descrição do projeto, se utilizaremos maven ou gradle, etc.

Para o nosso projeto ficará dessa maneira:



Após, definir as configurações basta clicar em Next.

A próxima tela vamos definir qual versão do spring utilizaremos e quais dependências.



Dependência utilizadas:

- **Spring Boot Dev Tools** - Um conjunto de ferramentas que ajudam no processo de desenvolvimento, uma delas é o restart automático, toda vez que um arquivo alterado é salvo a aplicação é reiniciada automaticamente.
- **Spring Web** - Usado para criar aplicações web inclusive RESTful, usando Spring MVC. Ele contém Tomcat embutido como container padrão.
- **Spring Data JPA** - Ele facilita o uso de tecnologias de acesso a dados, como banco de dados relacionais e não relacionais, etc.
- **H2 Database** - Este será nosso banco de dados ,ele é um banco de dados em memória, os dados se mantém somente enquanto a aplicação estiver rodando. Vamos utilizar ele pois pulamos alguns passos de instalação e configuração que precisaríamos fazer caso utilizássemos um banco de dados como oracle ou mysql.
- **Lombok** - Gera métodos getters e setters, equals, construtor completo e muitas outras coisas apenas com algumas anotações, deixando as classes mais limpas. Além da dependência é necessário instalar o plugin na IDE.

Organização do projeto

- **Controller** - Neste pacote ficarão os controllers de nossa api, os controllers devem ser responsáveis apenas por receber uma requisição e devolver uma resposta, neles iremos mapear as rotas de nossa api.
- **Dto** - Neste pacote ficarão os dtos de nossa api, DTO(Data Transfer Object) é um padrão de projeto usado para transporte de dados, na nossa api receberemos e devolveremos DTOs, isso porque não queremos expor nossas classes de modelo ao cliente.
- **Model** - Neste pacote ficarão nossas classes de modelos, cada classes representará uma tabela no nosso banco de dados.
- **Repository** - Neste pacote ficarão as classes responsáveis pela lógica de acesso a dados.
- **Service** - Neste pacote ficarão as classes responsáveis pelas regras de negócio da nossa aplicação, essas classes farão a ponte entre o controller e o model.
- **Utils** - Nesse pacote ficarão algumas classes utilitárias para nossa aplicação.

Iniciando o Desenvolvimento

Vamos começar criando o nosso controller, criaremos um ApostaController, ele conterá os endpoints da nossa API.

ApostaController.java

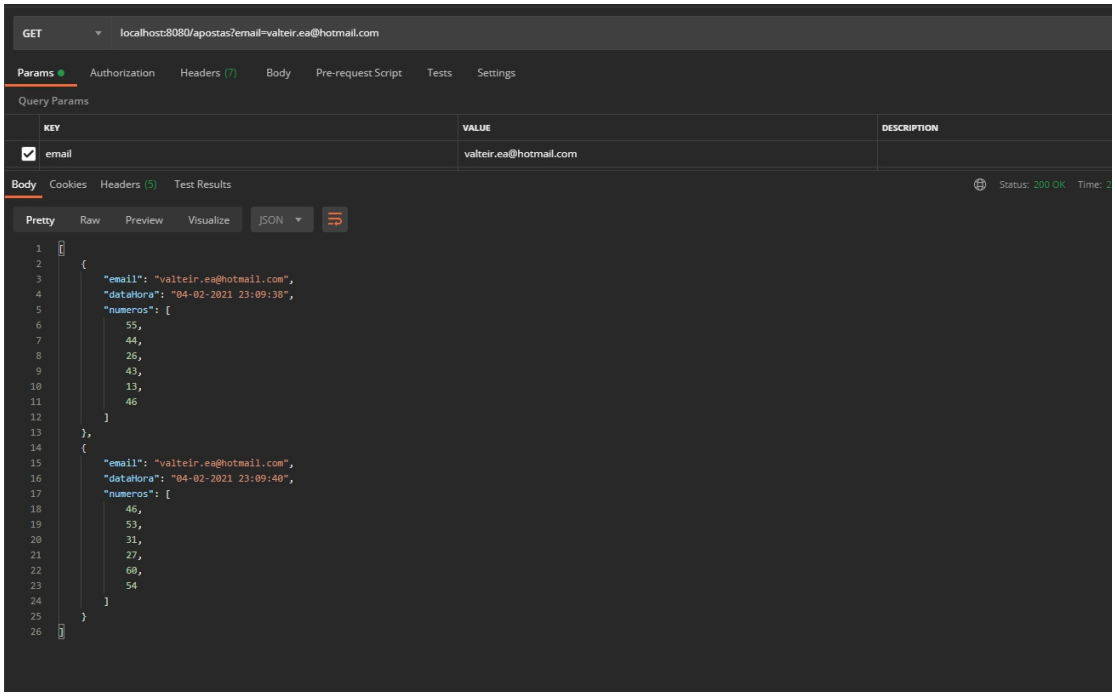
```java
package com.zup.loto.controller;

import java.util.List;

@RestController
@RequestMapping("/apostas")
public class ApostaController {

    @Autowired
    private ApostaService apostaService;

    @GetMapping
    public ResponseEntity<List<ApostaResponseDTO>> listarApostas(@RequestParam String email) {
        List<ApostaResponseDTO> apostas = apostaService.listarApostas(email);

        return ResponseEntity
                .status(HttpStatus.OK)
                .body(apostas);
    }

    @PostMapping
    public ResponseEntity<ApostaResponseDTO> apostar(@RequestBody ApostaRequestDTO apostaRequestDTO) {
        ApostaResponseDTO aposta = apostaService.apostar(apostaRequestDTO);

        return ResponseEntity
                .status(HttpStatus.CREATED)
                .body(aposta);
    }

}
```

Usamos a anotação @RestController para dizer que nossa classe é um serviço REST e usamos também a anotação @RequestMapping para mapear as requisições web para nosso controller.

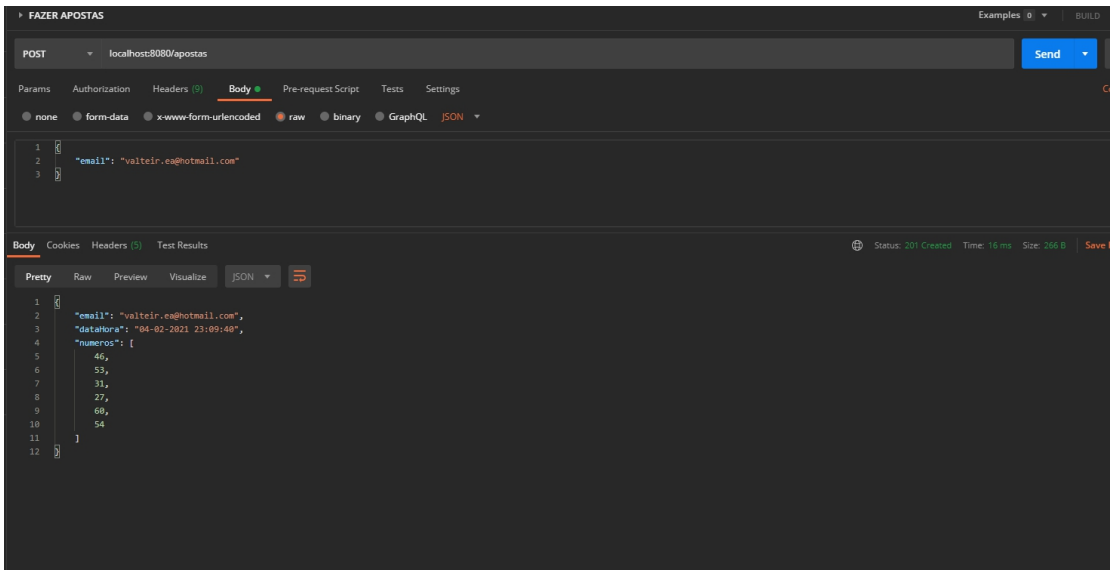
Nosso controller possuí um atributo do tipo ApostaService com a anotação @Autowired, essa anotação é utilizada para o Spring fazer a injeção de dependência.

ApostaController possuí dois métodos, apostar e listarApostas, o método listarApostas recebe um email como parâmetro e devolve uma lista das apostas feitas utilizando aquele email, esse método é anotado com @GetMapping o que significa que toda requisição do tipo GET em /apostas será direcionada para esse método. Caso a requisição seja realizada com sucesso o status code retornado será 200 OK

Vejamos como ficaria uma requisição GET para /apostas utilizando o postman:



O método apostar é utilizado para realizar uma nova aposta, ele recebe um objeto que contenha um atributo email e devolve a informação da aposta contendo os números apostados, esse método é anotado com @PostMapping isso significa que toda requisição do tipo POST em /apostas será direcionada para ele. Caso a requisição seja realizada com sucesso o status code retornado será 201 CREATED.



Nosso controlador recebe e devolve DTOs, esses são os dois DTOs utilizados:

ApostaRequestDTO.java

```java
package com.zup.loto.dto.request;

import lombok.Data;

@Data
public class ApostaRequestDTO {
    private String email;
}

```

ApostaResponseDTO.java

```java
package com.zup.loto.dto.response;

import java.time.LocalDateTime;

@Data
@Builder
public class ApostaResponseDTO {

    private String email;

    @JsonFormat(pattern = "dd-MM-yyyy HH:mm:ss")
    private LocalDateTime dataHora;

    private List<Integer> numeros;


}

```

Note que essas duas classes estão usando anotações do lombok, servem para omitir getters e setter principalmente entre outras coisas, deixando nossa classe menos poluída.

Como puderam ver nosso controller é responsável apenas por receber requisições e devolver respostas, agora vamos dar uma olhada na classe ApostaService pois ela esta cuidando das regras de negócio de nossa API.

ApostaService.java

```java
@Service
public class ApostaService {

    @Autowired
    private ApostaRepository apostaRepository;

    @Autowired
    private ApostaNumeroRepository apostaNumeroRepository;

    @Transactional
    public ApostaResponseDTO apostar(ApostaRequestDTO apostaRequestDTO) {
        Aposta aposta = new Aposta(apostaRequestDTO.getEmail());
        aposta = apostaRepository.save(aposta);

        List<ApostaNumero> numeros = gerarNumerosAposta();

        for(ApostaNumero num : numeros) {
            num.setAposta(aposta);
        }

        apostaNumeroRepository.saveAll(numeros);

        aposta.setNumeros(numeros);

        return entidadeParaResponseDTO(aposta);
    }

    @Transactional
    public List<ApostaResponseDTO> listarApostas(final String email) {
        List<Aposta> apostas = apostaRepository.findByEmailOrderByDataHora(email);

        List<ApostaResponseDTO> apostasResponseDTO = new ArrayList<>();

        apostas.forEach(ap -> apostasResponseDTO.add(entidadeParaResponseDTO(ap)));

        return apostasResponseDTO;
    }

    private List<ApostaNumero> gerarNumerosAposta() {
        RandomUtil gerador = new RandomUtil();
        List<ApostaNumero> numeros = new ArrayList<>();

        gerador.gerarNumeros().forEach(n -> {
            ApostaNumero num = new ApostaNumero();
            num.setNumero(n);
            numeros.add(num);
        });

        return numeros;
    }

    private ApostaResponseDTO entidadeParaResponseDTO(final Aposta aposta) {
        return ApostaResponseDTO.builder()
                .email(aposta.getEmail())
                .dataHora(aposta.getDataHora())
                .numeros(aposta.getNumeros().stream().map(n -> n.getNumero()).collect(Collectors.toList()))
                .build();
    }
}
```

A classe ApostaService possui dois atributos, ApostaRepository e ApostaNumeroRepository, eles são responsáveis pelo acesso aos dados de nossa aplicação.

ApostaNumeroRepository.java

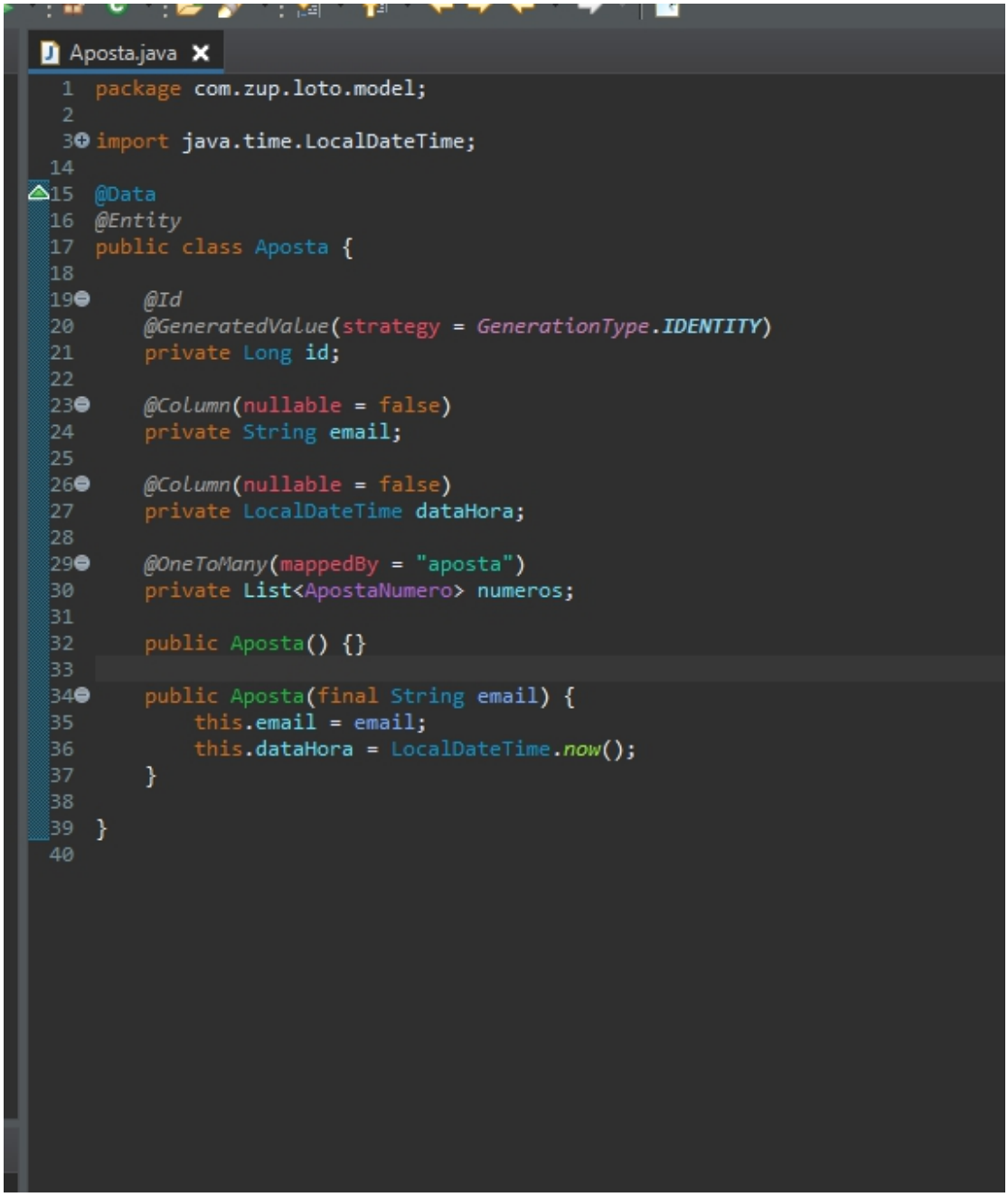
```java
package com.zup.loto.repository;

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

public interface ApostaNumeroRepository extends JpaRepository<ApostaNumero, Long> {

}
```

ApostaRepository.java

```java
package com.zup.loto.repository;

import java.util.List;

public interface ApostaRepository extends JpaRepository<Aposta, Long> {

    List<Aposta> findByEmailOrderByDataHora(String email);
}
```

As duas são interfaces e devem estender JpaRepository, repare que a ApostaRepository possuí um método que adicionamos, se seguirmos determinado padrão de escrita conseguimos criar consultas personalizadas como essa findByEmailOrderByDataHora esse método busca todas as apostas de um determinado email ordenando por data e hora. O spring fornece a implementação em tempo de execução.

No método apostar da classe ApostaService recebemos um objeto ApostaRequestDTO que possui o email do solicitante, a partir desse email criamos um objeto Aposta que representa nossa tabela no banco de dados.



```java
package com.zup.loto.model;

import java.time.LocalDateTime;

@Data
@Entity
public class Aposta {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private LocalDateTime dataHora;

    @OneToMany(mappedBy = "aposta")
    private List<ApostaNumero> numeros;

    public Aposta() {}

    public Aposta(final String email) {
        this.email = email;
        this.dataHora = LocalDateTime.now();
    }

}
```

@Entity para dizer que é uma entidade do banco de dados, @Id para informar que é a chave primaria da tabela, @Column com nullable = false para dizer que não pode ser nulo e @OneToMany para especificar o tipo de relacionamento.

Após criarmos a aposta usamos o apostaRepository para salvar no banco de dados, depois chamamos o método gerarNumerosAposta para gerar os números. Com os números gerados vincularemos eles a nossa aposta e depois usamos o apostaNumeroReposity para salvar no banco de dados.

ApostaNumero.java

```java
package com.zup.loto.model;

import javax.persistence.Column;

@Data
@Entity
public class ApostaNumero {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private int numero;

    @ManyToOne
    @JoinColumn(name="aposta_id", nullable=false)
    private Aposta aposta;

}
```

Após salvar no banco de dados precisamos retornar para o nosso controller um ApostaResponseDTO, então usamos o método entidadeParaResponseDTO para converter nosso objeto Aposta para ApostaResponseDTO.

O método listarApostas funciona de forma semelhante, recebe um email como parâmetro, utiliza o repository para buscar as apostas filtrando pelo email e depois converte as apostas encontradas para DTO e devolve para o controller.

E assim finalizamos nossa API, claro que ainda tem diversas coisas que podem ser adicionadas para melhorar o projeto como adicionar validações com o BeanValidation, tratar exceções e retornar mensagens de erros personalizadas, poderíamos criar uma documentação com o Swagger, autenticação, adicionar logs e até mesmo implementar testes, mas isso estenderia muito esse documento.

O código completo se encontra em https://github.com/ValteirEleuterio/lotozup-api