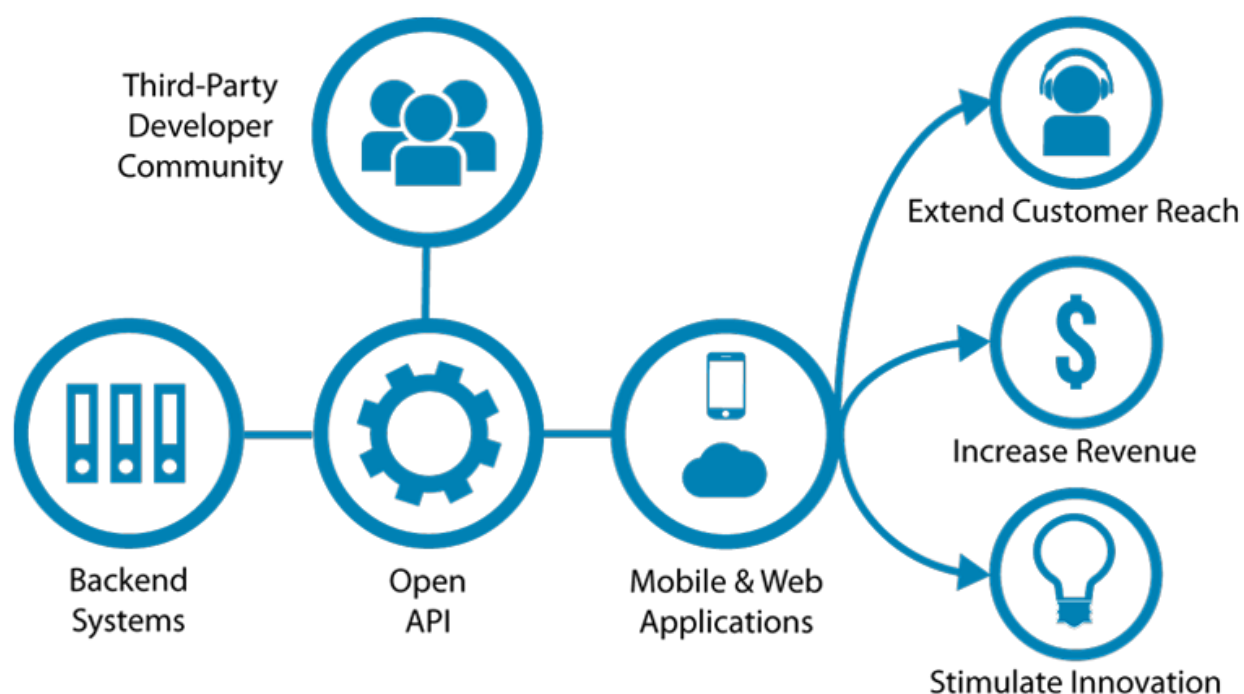


CONSTRUÇÃO DE UMA API REST PARA GERENCIAMENTO DE SORTEIOS

25 de janeiro de 2021



Tecnologias utilizadas:

Olá amigos, boa tarde! :)

Me chamo Lucas, sou estudante de programação. No post de hoje vamos construir, juntos, uma RestAPI (aplicação Web nos padrões Rest) para o desafio do Orange Talents da Zup ;) Espero ser bem assertivo e descontraído. O objetivo dessa brincadeira é explicar o conceito funcional de cada uma das tecnologias que vamos utilizar. Por hora nada muito aprofundado, trabalharemos de modo que qualquer pessoa, conhecedores ou não de linguagem de programação, possam desenvolver em conjunto com o Post. Agora, sem mais delongas, bora pra ação! Lembrando que vai ficar disponível todo código fonte do que será mostrado neste breve artigo;

Ferramentas que vamos utilizar: Spring Boot, Spring Data JPA, Spring Web, Spring devtools, validation, Spring Security, MySQL Driver, Maven, Swaeger.

BREVE EXPLICAÇÃO DAS FERRAMENTAS:

O Spring framework é um projeto do ecossistema. Resumidamente, as ferramentas que o Spring disponibiliza facilitam (e muito) a vida dos programadores Web. Ao desenvolver um projeto com o framework, conseguimos otimizar muito o tempo de trabalho, pois ele faz todo o preparo do ambiente de desenvolvimento, de modo a nós, desenvolvedores nos preocuparmos apenas com as regras de negócios (o que nossa aplicação faz, pra quem ela faz e quais seus requisitos de funcionamento). A seguir, vou dar uma explicação bem intuitiva de qual o papel de cada uma das tecnologias que vamos utilizar:

SpringBoot: o spring boot é a tecnologia do mundo Spring que monta todo o ambiente de desenvolvimento. Em resumo, ela faz todas as configurações necessárias para as demais tecnologias que utilizamos em um projeto Spring funcionarem. Além de claro, gerar um arquivo prontinho para importarmos e só nos preocuparmos com a modelagem do desenvolvimento da nossa aplicação (O arquivo gerado ainda traz o Maven instalado, tecnologias que falaremos mais a frente).

Spring Data JPA: funciona como uma interface que facilita a persistência (implementação e gerenciamento) de dados em um projeto Spring. Essa tecnologia nos traz vários métodos de comunicação com um banco de dados, como por exemplo métodos prontos para cadastrarmos e listarmos dados em uma tabela do banco. Maravilha, né? MySQL Driver: esse é o driver para utilização do MySQL em nossa aplicação. Existem diversos outros drivers disponíveis para conexão com várias outras bases.

Spring Web: este projeto Spring facilita o desenvolvimento para web. Traz várias ferramentas necessárias para padronização de uma aplicação com os modelos Rest, em conjunto com o Spring MVC (essa tecnologia já vem embutida quando inserimos a dependência Spring Web), que nos ajuda a desenvolver projetos seguindo o padrão de projeto MVC (Model, View e Controller). O padrão MVC em resumo é um projeto que possui o intuito de dividir nosso projeto em 3 partes: Model - responsável por toda parte de comunicação, inserção e retorno do banco de dados; View - toda parte visual da aplicação (tudo o que interage com o cliente); Controller - responsável pela comunicação entre a Model e a View, recebe as requisições da View, manda para a Model, e devolve para a View a resposta da Model.

Spring devTools: um dos grandes parceiros do desenvolvedor, essa tecnologia é muito útil e otimiza o ambiente de desenvolvimento. Um bom exemplo de sua aplicação: com Spring devtools instalado, não precisamos toda vez que alteramos algo em uma classe, “startar” novamente a aplicação. Top demais.

SpringValidation: fornece anotações responsáveis por realizar validações nos campos da nossa base de dados. Extremamente importante, pois funciona como uma segunda barreira de validação dos nossos dados. Normalmente a blindagem do tipo de dados que nosso back-end recebe acontece no front-end (para saber mais sobre front e back-end), mas as anotações validation constroem uma camada de segurança que não permite que dados indesejados sejam cadastrados no nosso banco de dados, caso a primeira camada de validação do front tenha alguma vulnerabilidade.

SpringSecurity: faz todo o trabalho de segurança da nossa aplicação. Aplica todas as camadas de segurança, não permitindo que um usuário faça alterações ou inserções em nossas bases de dados (podemos definir outras regras de negócio) se não estiver logado e possuir um token de segurança.

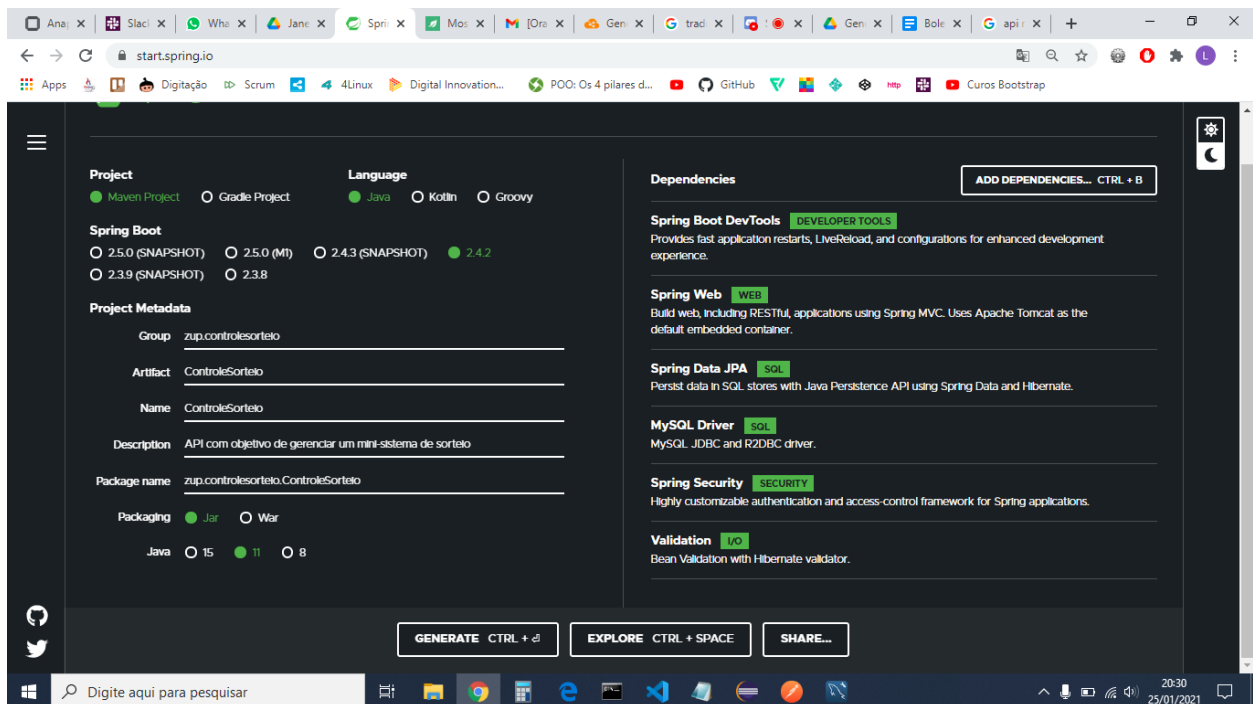
Maven: é o melhor amigo do desenvolvedor. Em poucas palavras, o Maven é o nosso gerenciador de dependências do projeto. Basicamente, ele gerencia, no nosso caso, todas as tecnologias que citamos acima, para que não haja conflito entre as versões das tecnologias que usamos.

Swagger: uma biblioteca da linguagem Java que gera automaticamente a documentação da nossa aplicação. :o fala aí, uma mão na roda.

Nossa API:

Vou chamar nosso mini-projeto de Controle de Sorteios. Para começar vamos gerar o arquivo no site do Spring Initializr. Vamos adicionar nas dependências todas as tecnologias que vamos utilizar, menos o Security (vamos implementá-lo mais a frente).

A tela do Initializr deve ficar algo parecido com isso:



Vamos criar todas as packages que vamos utilizar no projeto:

model; controller; repository; service; config e segurança

Vamos criar todas as classes que vamos utilizar no projeto:

Nossas classes que ficaram em nossa package model serão: Apostas, Cliente e futuramente Usuario e UsuarioLogin (para nossa camada de segurança);

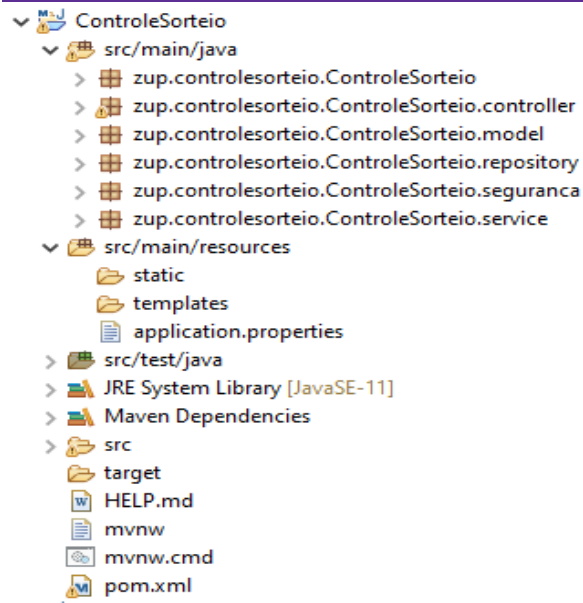
Nossas classes que ficaram em nossa package controller serão: ApostasController, ClienteController e futuramente UsuarioController (segurança);

Nossas classes que ficaram em nossa package repository serão: ApostasRepository, ClienteRepository e futuramente UsuarioRepository;

Nossas classes que ficaram em nossa package service serão: RegrasNegocio e futuramente UsuarioService;

*No final da nossa aplicação, vamos implementar a biblioteca Swagger, para gerar a documentação dos endpoints da nossa aplicação.

O projeto deve estar parecido com isso:



*A IDE que estou utilizando para desenvolvimento é o Eclipse, mas isso não impede que esse mesmo projeto seja desenvolvido em qualquer outra IDE da preferência do leitor.

Vamos começar criando nossa conexão com o banco de dados. No caso, vamos utilizar o MySQL. Primeiramente, devemos abrir o arquivo Application.properties.

Dentro do arquivo vamos escrever os seguintes códigos para conexão:

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/db_controlesorteio?createDatabaseIfNotExist=true&serverTimezone=UTC&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.show-sql=true
```

A primeira linha diz respeito a o que o Spring Jpa deve fazer na sua base de dados toda vez que “Startarmos” a aplicação, no caso dizemos que ele deve dar um update, e basicamente o que ele faz é mapear as nossas entidades em nosso código e ver se todas estão espelhadas no nosso banco. Na segunda linha referenciamos o local onde o banco está funcionando, no caso localhost:3306, e ainda dizemos o nome do banco, com a condição de criá-lo, caso ele ainda não exista. Terceira e quarta linha dizem

respeito a usuário e senha para acesso ao banco. Resumidamente, a anotação `show-sql = true` serve para conseguirmos ver os códigos SQL's nativos gerados durante os processos da nossa API.

Dando continuidade, agora vamos criar os modelos de entidades que serão os moldes para gerarmos nossas tabelas no banco de dados. Nossos modelos em primeiro momento serão as classes `Cliente` e `Apostas`, mas futuramente implementamos a classe `usuário`, necessária para segurança de nossa aplicação. Criaremos na class `Cliente` os atributos: `Id`, `email` e `apostas`. Tipagens `long`, `String` e `String`, respectivamente. Usaremos o modificador de acesso `private`, e logo em seguida vamos gerar os métodos para acesso `Getters` and `Setters`.

Vamos inserir no código as seguinte anotações: `@Entity`; `@GeneratedValue(strategy = GenerationType.IDENTITY)`, como na imagem a seguir:

```
@Entity
public class Apostas {

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

    @NotBlank
    private String email;
    @NotNull
    private int aposta;

    public Apostas(String email, int aposta) {
        this.email = email;
        this.aposta = aposta;
    }
    public Apostas() {

    }
    public long getId() {
        return id;
    }
}
```

Em resumo @Entity - Anotação que "fala" para o Spring Data Jpa que nossa classe é uma entidade e que deve ser criada uma tabela no banco com todos os atributos e tipagens referentes. @GeneratedValue - Anotação para definirmos qual atributo será nossa chave primária quando nossa tabela for criada na base de dados. Finalizamos a criação da Model Apostas. Vamos seguir os mesmos passos para criação da model Cliente, desta vez com os atributos: Id(long), email(String), nome(String), telefone(String) e numeroApostado(int). Id será nossa chave primária. Lembrando que é nesta etapa que acrescentamos as anotações validations. As mais comuns são @NotBlank (o campo não pode ser branco dentro do banco, isso significa que uma String do tipo espaço " " não é aceita no momento de cadastro) e @NotNull(o campo não pode ser nulo). Temos também a anotação validation @Size(determinamos o máximo e o mínimo de caracteres permitidos nessa coluna dentro do banco).

Vamos para nossa camada de repository. Criando ApostasRepository. Vamos criar uma interface com o nome ApostasRepository, e dentro da interface vamos estender outra interface, a do JPA, que é a interface responsável por nos dar todos os métodos prontos de relação com o banco. Vamos criar dentro da interface um método especial para utilizarmos na validação. Utilizaremos a notação @Query para conseguirmos realizar uma consulta que retorna uma String com os possíveis resultados. O código deve ficar como na imagem abaixo:

```
import java.util.List;

@Repository
public interface ApostasRepository extends JpaRepository<Apostas, Long>{

    @Query(value = "select aposta from apostas where email = :email", nativeQuery = true)
    public List<String> listarApostasPorEmail(@Param("email") String email);
}
```

Passamos como parâmetros para a interface ApostasRepository a classe em que vamos utilizar os métodos e a tipagem da chave primária, no exemplo, usamos a

classe "Apostas" que tem a tipagem da chave primária como long. Vamos seguir os mesmos passos para criação da ClienteRepository, criando também um método especial para seleção de email de cliente. Como a interface ClienteRepository deve ficar:

```
@Repository
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

    @Query(value="select id from cliente where email = :email", nativeQuery = true)
    public String buscaClientePorEmail(@Param("email") String email);
}
```

Partimos então para a criação das classes controller. Vamos criar primeiro a classe ApostasController. Com a classe criada, colocamos a anotação @RestController e @RequestMapping("/NomeParaAcessarmosOsEndpointsDeCadaEntidade") logo em cima da declaração da classe. Após isso, vamos criar um objeto do tipo ApostasRepository (através desse objeto vamos acessar os tão falados métodos de acesso a banco de dados(bd) do Spring Data JPA). Em cima da declaração, vamos colocar a anotação @Autowired. Essa anotação trabalha com o conceito de injeção de dependências, que basicamente "instância" esse objeto da ApostasRepository. Agora chegou o momento de criarmos os endpoints, para isso vamos usar as anotações @GetMapping, @PostMapping, @PutMapping e @DeleteMapping.

Para construir os endpoints, vamos trabalhar com o conceito de ResponseEntity<>, uma classe que vai "transformar" nossas respostas para o modelo Rest, devolvendo um Status de resposta. Para o método Get, precisamos publicar uma list do método ResponseEntity do tipo Cliente sem parâmetros, e que retorna um ResponseEntity com uma list do tipo Cliente, usando o método findAll() da interface ApostasRepository.

Temos um @GetMapping("/listar/{email}") adicional onde utilizamos o método especial de listagem criado em nosso ApostasRepository.

Para construir o Post, vamos receber como parametro o objeto Apostas vindo do corpo da página. Publicaremos um método ResponseEntity, do tipo String, que recebe o objeto apostas da body (para isso vamos usar a anotação @RequestBody antes do objeto para informar que ele vem do corpo da página), e retorna um ResponseEntity, do tipo String, com um status "Created" caso tudo dê certo, e retorna no corpo da página o objeto Apostas cadastrado.

O método Put segue a mesma arquitetura do método Post, porém dessa vez, ao contrário do método Post, vamos precisar enviar um objeto Apostas completo já existente na Body, mas com a chave primária Id, necessária para identificação e alteração das informações necessárias.

O método Delete é o mais simples ;) Para eles, vamos construir um método sem retorno (void) e passamos como parâmetro o Id da aposta que desejamos excluir (passamos o id pela URL da requisição). Para informarmos para o método que o parâmetro que estamos passando vem da URL usamos a anotação @PathVariable. E realizamos a ação de deleção do objeto que tem o id passado, através do método deleteById do objeto de ApostasRepository.

Vamos seguir todos os mesmos passos para criação da ClienteRepository, alterando os parâmetros e tipos, quando necessário.

* Para o ClienteController, precisamos criar também métodos especiais de validação de cadastro, mas veremos isso mais a frente na class RegrasNegocio.

Como devem ficar as classes ClienteController e ApostasController:

```

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

    @Autowired
    ApostasRepository apostasRepository;
    @Autowired
    RegrasNegocio regrasNegocio = new RegrasNegocio();

    @GetMapping
    public ResponseEntity<List<Apostas>> listarApostas(){
        return ResponseEntity.ok(apostasRepository.findAll());
    }

    @GetMapping("/listar/{email}")
    public ResponseEntity<List<String>> listarApostasPorEmail(@PathVariable String email){
        return ResponseEntity.ok(apostasRepository.listarApostasPorEmail(email));
    }

    @PostMapping("/cadastro")
    public ResponseEntity<String> cadastrarAposta(@RequestBody Apostas aposta){
        return regrasNegocio.cadastrarAposta(aposta);
    }

    @PutMapping("/atualizar")
    public ResponseEntity<Apostas> atualizarApostas(@RequestBody Apostas aposta){
        return ResponseEntity.ok(apostasRepository.save(aposta));
    }

    @DeleteMapping("/{id}")
    public void deletar(@PathVariable long id) {

```

```

@RestController
@RequestMapping("/cliente")
public class ClienteController {

    @Autowired
    ClienteRepository clienteRepository;
    @Autowired
    ApostasRepository apostaRepository;
    @Autowired
    RegrasNegocio tratativaDados = new RegrasNegocio();

    @GetMapping("/geradorsorteio")
    public ResponseEntity<String> geradorSorteio(){
        int numeroSorteado = tratativaDados.geradorSorteio();
        String sorteio = Integer.toString(numeroSorteado);
        return ResponseEntity.ok(sorteio);
    }

    @PostMapping("/cadastro")
    public ResponseEntity<String> cadastrarCliente(@RequestBody Cliente cliente){
        return tratativaDados.protecaoDadoscliente(cliente);
    }

    @PutMapping("/atualizar")
    public ResponseEntity<Cliente> atualizarCliente(@RequestBody Cliente cliente){
        return ResponseEntity.ok(clienteRepository.save(cliente));
    }

    @DeleteMapping
    public void deletar() {
        clienteRepository.deleteAll();
    }
}

```

Nesse momento vamos criar a classe que será a responsável pela nossa regra de negócios, a RegraNegocio. Essa classe não terá atributos, apenas métodos. Os métodos que publicaremos serão protecaoDadoscliente, geradorSorteio, cadastrarAposta, que terão como objetivo, proteger a nossa aplicação de cadastros de clientes com email duplicado, gerar números aleatórios para sorteio e proteção de cadastro de apostas para emails que não existam. Ambas serão ResponseEntity e terão tipagem String.

Resumidamente o que fazemos em cada um dos métodos:

protecaoDadosCliente - recebemos um objeto do tipo cliente e em seguida passamos o email desse cliente como parâmetro para o método especial que criamos em clienteRepository. Recebemos a resposta deste método em uma String e verificamos,

se ela for nula, significa que não temos este email cadastrado em nossa base, sendo assim, podemos cadastrar nosso cliente.

CadastrarAposta - segue a mesma lógica do método `protecaoDadosCliente`, porém desta vez se não existir um email na base, nós não cadastramos a aposta, pois não podemos cadastrar uma aposta sem cliente.

`geradorSorteio` - método sem parâmetro. Criamos um objeto do tipo `Random` e em seguida chamamos o método dessa classe que gera números aleatórios. O restante do código foi apenas para complementar a sequência de números para gerarmos uma aposta (essa etapa de geração de código poderia ser gerada no Front-end da aplicação, mas como esse artigo/post tem o intuito de demonstrar ferramentas back-end, construí esse método para auxiliar nos testes).

Assim ficou nossa classe `RegraNegocio`:

```
@Service
public class RegrasNegocio {

    @Autowired
    ClienteRepository clienteRepository;
    @Autowired
    ApostasRepository apostaRepository;

    public ResponseEntity<String> protecaoDadoscliente(Cliente cliente){
        String verifica = clienteRepository.bucaClientePorEmail(cliente.getEmail());
        if(verifica == null) {
            Apostas aposta = new Apostas(cliente.getEmail(), cliente.getNumeroApostado());
            apostaRepository.save(aposta);
            clienteRepository.save(cliente);
            return ResponseEntity.status(HttpStatus.CREATED).body("Cadastro realizado com sucesso!");
        }else {
            return ResponseEntity.ok("Já existe um usuário cadastrado com esse email!");
        }
    }
}
```

```

public int geradorSorteio() {
    Random aleatorio = new Random();
    int num = aleatorio.nextInt(10000), num2 = aleatorio.nextInt(20000), newNumero = 0;

    String numero = Integer.toString(num) + Integer.toString(num2);

    if(numero.length() < 10) {
        newNumero = Integer.parseInt(numero);

        while(newNumero < 1000000000) {
            newNumero = newNumero * 111;
        }
    }

    return newNumero;
}

public ResponseEntity<String> cadastrarAposta(Apostas aposta){
    String verifica = clienteRepository.bucaClientePorEmail(aposta.getEmail());

    if(verifica == null) {
        return ResponseEntity.ok("Aposta não pode ser cadastrada pois o email não existe!");
    }else {
        apostaRepository.save(aposta);
        return ResponseEntity.ok("Aposta cadastrada com sucesso!");
    }
}

```

Chegamos ao fim da implementação dos nossos end-points e nossas regras de negócios. Agora iremos implementar em nosso código a camada Security.

Em primeiro lugar vamos criar uma model Usuario e outra UsuarioLogin. A model Usuario será uma @Entity, a UsuarioLogin não.

Atributos da Usuario: Id (long - chave primário), String nome, String usuario, String senha;

Atributos da UsuarioLogin: nome (String), usuario (String), token(String), senha(String).

*A model UsuarioLogin será usada para autenticação e geração de token.

Como deverão ficar as models:

```
@Entity
public class Usuario {

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

    private String nome;

    private String usuario;

    private String senha;

    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 getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
}
```



```

public class UsuarioLogin {
    private String nome;
    private String usuario;
    private String senha;
    private String token;
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getSenha() {
        return senha;
    }
    public void setSenha(String senha) {
        this.senha = senha;
    }
    public String getToken() {
        return token;
    }
    public void setToken(String token) {
        this.token = token;
    }
}

```

Faremos agora a class UsuarioControler. Criaremos dois @PostMapping, onde usaremos os métodos Logar e Cadastrar, que criaremos mais a frente na class UsuarioService; O método Logar recebe como parâmetro um objeto do tipo Usuario, e passa como parâmetro para o método logar do objeto de UsuarioService, que por sua vez realiza toda regra de negócio e, se o usuário existir, devolve-nos o token para acessarmos os demais end-points. O método Cadastrar também recebe um objeto do tipo Usuario e realiza seu cadastro. Esses métodos são de extrema importância para segurança de uma aplicação. Não aprofundaremos muito no conteúdo sobre security, mas, a depender da aplicação, podemos criar nossa própria regra de negócios, como tipos de usuários e tipos de acesso.

```

@RestController
@RequestMapping("usuario")
public class UsuarioController {

    @Autowired
    UsuarioService usuarioService;

    @PostMapping("/cadastrar")
    public ResponseEntity<Usuario> cadastroUsuario(@RequestBody Usuario user){
        return ResponseEntity.status(HttpStatus.CREATED).body(usuarioService.CadastrarUsuario(user));
    }

    @PostMapping("/login")
    public ResponseEntity<UsuarioLogin> loginUsuario(@RequestBody Optional<UsuarioLogin> user){
        return usuarioService.login(user).map(resp -> ResponseEntity.ok(resp))
            .orElse(ResponseEntity.status(HttpStatus.UNAUTHORIZED).build());
    }
}

```

Para finalizar a implementação da camada de segurança e partir para realização de testes, seguem modelos de implementação da class UsuarioService, e de todas as classes da package Segurança:

UsuarioService:

```

@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repository;

    public Usuario CadastrarUsuario(Usuario usuario) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        String senhaEncoder = encoder.encode(usuario.getSenha());
        usuario.setSenha(senhaEncoder);

        return repository.save(usuario);
    }
}

```

```

public Optional<UsuarioLogin> login(Optional<UsuarioLogin> user){
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
    Optional<Usuario> usuario = repository.findByUsuario(user.get().getUsuario());

    if(usuario.isPresent()) {
        if(encoder.matches(user.get().getSenha(), usuario.get().getSenha())) {

            String auth = user.get().getUsuario() + ":" + user.get().getSenha();
            byte[] encodeAuth = Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")));
            String authHeader = "Basic " + new String(encodeAuth);

            user.get().setToken(authHeader);
            user.get().setNome(usuario.get().getNome());

            return user;
        }
    }

    return null;
}

```

BasicSecurityConfig:

```

@EnableWebSecurity
public class BasicSecurityConfig extends WebSecurityConfigurerAdapter{

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception{
        auth.userDetailsService(userDetailsService);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception{
        http.authorizeRequests()
            .antMatchers("/usuario/login").permitAll()
            .antMatchers("/usuario/cadastrar").permitAll()
            .anyRequest().authenticated()
            .and().httpBasic()
            .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and().cors()
            .and().csrf().disable();
    }
}

```

UserDetailsImpl:

```
@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repository;

    public Usuario CadastrarUsuario(Usuario usuario) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        String senhaEncoder = encoder.encode(usuario.getSenha());
        usuario.setSenha(senhaEncoder);

        return repository.save(usuario);
    }

    public Optional<UsuarioLogin> login(Optional<UsuarioLogin> user){
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
        Optional<Usuario> usuario = repository.findByUsuario(user.get().getUsuario());

        if(usuario.isPresent()) {
            if(encoder.matches(user.get().getSenha(), usuario.get().getSenha())) {

                String auth = user.get().getUsuario() + ":" + user.get().getSenha();
                byte[] encodeAuth = Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")));
                String authHeader = "Basic " + new String(encodeAuth);

                user.get().setToken(authHeader);
                user.get().setNome(usuario.get().getNome());

                return user;
            }
        }

        return null;
    }
}
```

UserDetailsServiceImpl:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UsuarioRepository userRepository;
































    @Override
    public UserDetails loadUserByUsername(String userName) throws UsernameNotFoundException {

        Optional<Usuario> user = userRepository.findByUsuario(userName);
        user.orElseThrow(() -> new UsernameNotFoundException(userName + " not found."));

        return user.map(UserDetailsImpl::new).get();
    }
}
```

TESTANDO A APLICAÇÃO:

Como deve ficar nossa aplicação no final:

- ▼  ControleSorteio
 - ▼  src/main/java
 - ▼  zup.controlesorteio.ControleSorteio
 - >  ControleSorteioApplication.java
 - ▼  zup.controlesorteio.ControleSorteio.controller
 - >  ApostasController.java
 - >  ClienteController.java
 - >  UsuarioController.java
 - ▼  zup.controlesorteio.ControleSorteio.model
 - >  Apostas.java
 - >  Cliente.java
 - >  Usuario.java
 - >  UsuarioLogin.java
 - >  zup.controlesorteio.ControleSorteio.repository
 - ▼  zup.controlesorteio.ControleSorteio.seguranca
 - >  BasicSecurityConfig.java
 - >  UserDetailsImpl.java
 - >  UserDetailsServiceImpl.java
 - ▼  zup.controlesorteio.ControleSorteio.service
 - >  RegrasNegocio.java
 - >  UsuarioService.java
 - >  src/main/resources
 - >  src/test/java
 - >  JRE System Library [JavaSE-11]
 - >  Maven Dependencies
 - >  src
 - >  target
 - >  HELP.md
 - >  mvnw
 - >  mvnw.cmd
 - >  pom.xml

Cadastrando usuário e logando:

The screenshot displays a REST client interface with two tabs at the top: `GET localhost:8080/cliente/geradors...` and `POST localhost:8080/usuario/cadast...`. The `POST` tab is active, showing the endpoint `localhost:8080/usuario/cadastrar`. The request body is a JSON object: `{ "nome": "Lucas Gato", "usuario": "LucasGato", "senha": "1234" }`. The response status is `201 Created` with a time of `509 ms` and a size of `561 B`. The response body is a JSON object: `{ "id": 3, "nome": "Lucas Gato", "usuario": "LucasGato", "senha": "$2a$10$.m9G7xkzbNYtjwFRyjVTiOfb6hwV3fBwN9B24knoHdj1syMeiR8h2" }`.

GET localhost:8080/cliente/geradors... POST localhost:8080/usuario/cadast... No Environment

Untitled Request BUILD

POST localhost:8080/usuario/cadastrar Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "nome": "Lucas Gato",
3   "usuario": "LucasGato",
4   "senha": "1234"
5 }
```

Body Cookies Headers (14) Test Results Status: 201 Created Time: 509 ms Size: 561 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 3,
3   "nome": "Lucas Gato",
4   "usuario": "LucasGato",
5   "senha": "$2a$10$.m9G7xkzbNYtjwFRyjVTiOfb6hwV3fBwN9B24knoHdj1syMeiR8h2"
6 }
```

GET localhost:8080/cliente/geradors... POST localhost:8080/usuario/logar + ... No Environment

Untitled Request BUILD

POST localhost:8080/usuario/logar Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

Headers 8 hidden

	KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	Basic THVjYXNHYXRvOjEyMzQ=				
	Key	Value	Description			

Body Cookies Headers (14) Test Results Status: 200 OK Time: 304 ms Size: 530 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "nome": "Lucas Gato",
3   "usuario": "LucasGato",
4   "senha": "1234",
5   "token": "Basic THVjYXNHYXRvOjEyMzQ="
6 }
```

Bootcamp

Gerando um número para sorteio:

GET localhost:8080/cliente/geradors... GET localhost:8080/cliente/geradors... + ... No Environment

Untitled Request BUILD

GET localhost:8080/cliente/geradorsorteio Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

Headers 8 hidden

	KEY	VALUE	DESCRIPTION	***	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Authorization	Basic THVjYXNHYXRvOjEyMzQ=				
	Key	Value	Description			

Body Cookies Headers (14) Test Results Status: 200 OK Time: 153 ms Size: 445 B Save Response

Pretty Raw Preview Visualize Text

```
1 1368597450
```

Cadastrando um cliente:

GET localhost:8080/cliente/geradors... POST localhost:8080/cliente/cadastro

Untitled Request

POST localhost:8080/cliente/cadastro

Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": "lucas2.gato.03092001@gmail.com",
3   "nome": "Lucas Gato",
4   "telefone": "(19)9.8768-0434",
5   "numeroApostado": 1368597450
6 }
```

Body Cookies Headers (14) Test Results

Status: 201 Created Time: 254 ms Size: 471 B Save Response

Pretty Raw Preview Visualize Text

1 Cadastro realizado com sucesso!

Se tentar cadastrar novamente:

GET localhost:8080/cliente/geradors... POST localhost:8080/cliente/cadastro

Untitled Request

POST localhost:8080/cliente/cadastro

Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": "lucas2.gato.03092001@gmail.com",
3   "nome": "Lucas Gato",
4   "telefone": "(19)9.8768-0434",
5   "numeroApostado": 1368597450
6 }
```

Body Cookies Headers (14) Test Results

Status: 200 OK Time: 157 ms Size: 484 B Save Response

Pretty Raw Preview Visualize Text

1 Já existe um usuário cadastrado com esse email!

Cadastrando uma aposta para cliente não existente:

GET localhost:8080/cliente/geradors... POST localhost:8080/apostas/cadast...

Untitled Request BUILD

POST localhost:8080/apostas/cadastro Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": "lucas2sa.gato.03092001@gmail.com",
3   "numeroApostado": 1368597450
4 }
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 222 ms Size: 492 B Save Response

Pretty Raw Preview Visualize Text

1 Aposta não pode ser cadastrada pois o email não existe!

Bootcamp

Cadastro de uma aposta para um cliente:

GET localhost:8080/cliente/geradors... POST localhost:8080/apostas/cadast...

Untitled Request BUILD

POST localhost:8080/apostas/cadastro Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "email": "lucas2.gato.03092001@gmail.com",
3   "numeroApostado": 1368597450
4 }
```

Body Cookies Headers (14) Test Results Status: 200 OK Time: 157 ms Size: 465 B Save Response

Pretty Raw Preview Visualize Text

1 Aposta cadastrada com sucesso!

Bootcamp

Adicionando Swagger para documentação:

Para adicionar o Swagger em nosso código para geração de documentação, precisamos acrescentar as seguintes dependências em nosso arquivo maven de gerenciamento de dependências, o pom.xml (último arquivo do nosso projeto):

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
</dependency>
```

Agora vamos criar uma classe SwaggerConfig, em uma package chamada config.

Na SwaggerConfig vamos colocar os seguintes códigos:

```

ableSwagger2
nfiguration
lic class SwaggerConfig {
    @Bean
    public Docket docket() {
        return new Docket(DocumentationType.SWAGGER_2).select()
            .apis(RequestHandlerSelectors.basePackage("zup.controlesorteio.ControleSorteio.controller"))
            .paths(PathSelectors.any()).build().apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("Gerenciador de sorteio").description("API de gerenciamento de sorteio")
            .contact(contact()).build();
    }

    private Contact contact() {
        return new Contact("Lucas Gabriel", "https://github.com/lucas-ggbriel",
            "Estudante de programação Java Full-Stack");
    }
}

```

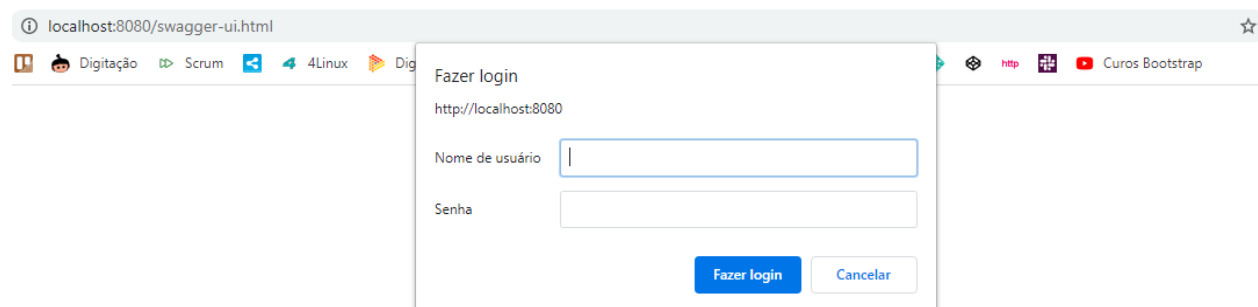
em `.basepackage("")` colocamos a package em que se encontram nossos endpoints para geração de documentação.

em `.title()` colocamos o título do nosso projeto.

em `.description()` colocamos a descrição do projeto e em `contact()` colocamos todas nossas informações de contato, como nome, link do git ou outro link de preferência e também até uma breve descrição.

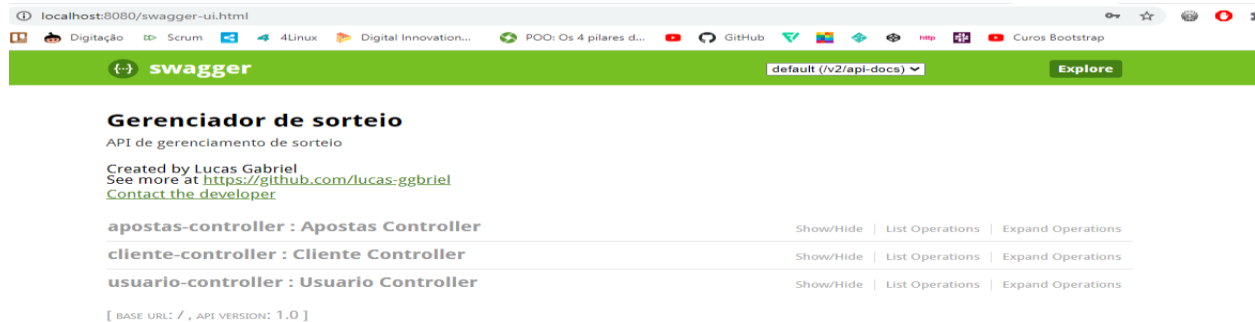
Acessando o Swagger:

Após inicializar a aplicação, caso ela ainda esteja rodando em servidor local, basta acessar o link `http://localhost:8080/swagger-ui.html`.



Ao tentarmos acessar a documentação, por nossa API possuir uma camada de segurança, a aplicação vai pedir para entrarmos. Basta colocar o login e senha cadastrados.

Após entrarmos encontramos toda a documentação de nossa aplicação:



Finalizamos a aplicação. Algumas considerações:

A tabela cliente e apostas podem ser relacionadas, porém como a aplicação teve o intuito de conhecer um pouco melhor as tecnologias do mundo Spring e como aplicá-las em um projeto real, não nos atentamos muito a isso. Mas é uma ótima implementação para uma versão 2.0 da aplicação.

Para realização do Deploy da API, temos diversas opções. Podemos fazê-lo a nível local, com o uso de ferramentas como o Docker, um poderoso gerenciador de containers, e também sites de hospedagem gratuitos como o Hiroku. Neste artigo/post não vamos fazer o deploy da aplicação para evitar delongas. Mas é também uma ótima implementação para a versão 2.0 dessa aplicação!

Muito obrigado pela atenção :)

Link do projeto no GitHub: <https://github.com/lucas-ggbriel/Desafio-Zup.git>