Desafio Zup Orange Talents

Autor: Guilherme Gwadera

Artigo escrito para o processo seletivo Orange Talents. Caso deseje acessar o repositório com o código-fonte ou rodar a aplicação, por gentileza vá para a última seção para mais detalhes.

Sumário

- Desafio Zup Orange Talents
  - Sumário
  - 1. Introdução
  - 2. Iniciando o projeto
  - 3. Definindo as entidades
  - 4. Conversando com o banco de dados
  - 5. A lógica do negócio
  - 6. Recebendo as requisições
    - Criando uma conta
    - Acessando uma conta
  - 7. Tratando os erros
  - 8. Rodando a aplicação
  - 9. Concluindo

1. Introdução

Neste artigo será descrito o desenvolvimento de uma API REST para a criação de usuários para uma aplicação bancária. A API será implementada utilizando a linguagem Java com os frameworks Spring + Hibernate.

Resumidamente, a API receberá as informações de nome, email, CPF e data de nascimento, necessários para a abertura de uma nova conta de usuário, e a aplicação deverá então validar e armazenar a conta em um banco de dados relacional. Será necessário também responder à requisição com o código de status adequado e demais informações necessárias.

A utilização dos frameworks citados facilita muito o desenvolvimento da API, abstraindo grande parte do trabalho necessário para implementar as camadas web HTTP, fundamentais para os serviços REST, e a comunicação com o banco de dados relacional. O módulo *Spring Boot* incluso no framework também auxilia com a configuração dos *beans* (componentes) da aplicação, permitindo que praticamente todas as configurações sejam feitas por anotações no código, sem precisar lidar com arquivos XML.

2. Iniciando o projeto

O primeiro passo para qualquer projeto Spring é gerar os arquivos bases de configurações e dependências. Essa etapa pode ser feita através do site Spring Initializr, por onde você define os metadados do projeto e seleciona as dependências desejadas.

Para esta aplicação foi escolhido a versão 15 do Java por ser a mais recente, e o *Gradle* como gerenciador do projeto por possuir uma interface e configuração mais simples em relação ao

Maven. As dependências selecionadas foram as seguintes:

- Spring Boot DevTools: ferramentas que auxiliam no desenvolvimento de projetos Spring, como, por exemplo, o hot reload (recarregar a aplicação sem reinicia-la após mudanças).
- Lombok: gerador de código como construtores, getter e setters para classes, para acelerar e facilitar o desenvolvimento.
- Spring Web: fornece ferramentas para a criação dos serviços web, e também inclui o servidor HTTP Apache Tomcat.
- Spring Data JPA: ferramentas para a comunicação com banco de dados utilizando o Java Persistence API (JPA) e Hibernate.
- H2 Database: banco de dados relacional em memória, suficiente para o escopo simples da aplicação. Em um caso real, seria mais adequado utilizar outro banco de dados como o MySQL, PostgreSQL, MariaDB, etc.
- Validation: fornece anotações que facilitam a validação dos dados recebidos pela API.

Uma cópia dessas configurações pode ser obtida através deste link .

Finalizado a configuração e seleção de dependências, basta clicar no botão *Generate* que será gerado e baixado um arquivo contendo tudo que o projeto Spring precisa. Após extrair o conteúdo do arquivo, o projeto pode ser importado em alguma IDE, como o Intellij IDEA ou Eclipse. Após a importação, você verá que o projeto possui somente o wrapper do Gradle, necessário para rodar o projeto, e um pacote dentro da pasta `src/main/java`, contendo uma única classe nesse caso chamada de `ChallengeApplication`, com o código abaixo.

```
@SpringBootApplication
public class ChallengeApplication {

    public static void main(String[] args) {
        SpringApplication.run(ChallengeApplication.class, args);
    }

}
```

Essa classe é responsável por iniciar a aplicação e todos os seus componentes. A anotação `@SpringBootApplication` é um conjunto de várias outras anotações que habilitam a configuração automática de todos os componentes Spring do projeto através do Spring Boot.

3. Definindo as entidades

Com a base do projeto já gerada e importada na sua IDE preferida, podemos agora colocar a mão na massa e começar a construir a nossa aplicação. O desenvolvimento da API pode começar por qualquer uma das camadas, seja os modelos, serviços, ou o controladores para as requisições. Porém, neste caso foi optado por começar pela definição do modelo para a conta que será criada.

Sabemos que para a conta de usuário na aplicação bancária teremos os seguintes campos:

- Nome
- Email
- CPF
- Data de nascimento

Não foi explicitamente especificado se todos os campos são obrigatórios, mas pensando no contexto de uma conta bancária, é assumido que os 4 campos serão necessários durante o registro. Além disso, será necessário também fazer a validação de campos como o email e CPF, e opcionalmente verificar se ambos já existem no banco de dados.

Com isso em mente, podemos escrever uma definição para a nossa classe de usuário:

```
package talents.orange.challenge.model;

/* imports omitidos */

@Data // Lombok: gerar getters, setters, toString, etc
@NoArgsConstructor // Lombok: gerar construtor vazio
@AllArgsConstructor // Lombok: gerar construtor com todos os campos
@Builder // Lombok: gera construtor com o design pattern de Builder
@Entity // JPA: define que a classe será uma entidade a ser persistida
public class User implements Serializable {

    @Serial
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @JsonIgnore // ignora esse campo na serialização
    private Long id;

    @NotBlank
    @Column(nullable = false)
    private String name;

    @Email // validar email
    @NotBlank
    @Column(unique = true, nullable = false)
    private String email;

    @CPF // validar CPF
    @NotBlank
    @Column(unique = true, nullable = false)
    private String cpf;
```

```
    @Past // validar se a data é no passado
    @NotNull
    @Column(nullable = false)
    private LocalDate birthday;

}
```

Com o código acima estamos criando o modelo da conta de usuário para a aplicação, definindo os 4 campos citados acima, e também um campo de ID gerado automaticamente pelo banco de dados. Pode-se observar que com a ajuda do Lombok grande parte do código como os getters, setters e construtores são gerados automaticamente, deixando somente o que realmente importa para a lógica do software.

Além disso, também foram adicionadas as anotações de validação, como `@NotBlank` para evitar strings vazias ou nulas, `@Past` que valida se a data está obrigatoriamente no passado, `@Email` para validar que o email enviado é está em um formato válido, e `@CPF` que valida o CPF conforme as regras da Receita Federal. Outro ponto importante é que foi optador por armazenar o CPF como string ao invés de um formato numérico, como `long`, para evitar problemas caso o CPF comece com 0, fazendo que os dígitos iniciais sejam perdidos.

4. Conversando com o banco de dados

De forma a persistir as contas de usuários em um banco de dados, é preciso que a nossa aplicação se comunique de alguma forma com o software do BD. Para isso é utilizado o *Hibernate*, incluso no pacote *Spring Data JPA*. O *Hibernate* é uma implementação das regras definidas pelo *Java Persistence API* (JPA), que descreve uma API comum para os frameworks de persistência de dados, mapeando os objetos Java (POJO — *Plain Old Java Objects*) para entidades relacionais.

O mapeamento destes objetos para o banco de dados também pode ser feito de várias formas, desde a mais complexa, declarando passo a passo desde à conexão com o banco usando o JDBC, ou utilizando as abstrações fornecidas pelo framework Spring. Felizmente essas abstrações facilitam bastante este trabalho.

Como é possível ver pelo código a seguir, basta definir uma interface de repositório que estende a interface `JpaRepository`, fornecida pelo módulo Spring Data. Feito isso, automaticamente o repositório já possuirá um conjunto de CRUD completo gerado ao compilar o projeto, podendo adicionar, editar, remover e buscar entidades. A anotação `@Repository` diz para o Spring que estamos declarando um novo componente de repositório, que será então incluído durante a configuração automática do Spring Boot e que estará disponível para a injeção de dependência.

```
package talents.orange.challenge.repository;

/* imports omitidos */

@Repository
public interface UserRepository extends JpaRepository<User, Long> {}
```

Agora pensando um pouco à frente, vamos precisar de mais dois métodos que interagem com o banco de dados para verificar se já existe algum email ou CPF salvos. Nesse quesito a abstração do framework também é de grande ajuda, pois somente declarando um método com a operação e nome do campo, a implementação já é gerada automaticamente. Assim, declarando os métodos `existsByCpf` e `existsByEmail` finalizamos o nosso repositório, possibilitando a validação completa durante o registro.

```
package talents.orange.challenge.repository;

/* imports omitidos */

@Repository // declara como componente de repositório
public interface UserRepository extends JpaRepository<User, Long> {

    boolean existsByCpf(String cpf);

    boolean existsByEmail(String email);

}
```

5. A lógica do negócio

Agora com o modelo e repositório prontos, podemos partir para construir a lógica de negócio da nossa aplicação. Os requisitos são simples: receber as informações da nova conta, validar, verificar se é duplicado, e persistir no banco de dados. A parte de receber os dados e validar será feita um pouco adiante, primeiramente vamos focar em programar a verificação de duplicidade e a persistência da entidade.

Em projetos Spring, é comum programar as lógicas de negócio em um componente de serviço, que também utilizará os componentes de repositório para interagir com o banco de dados. Os métodos desse componente serão então chamados dentro do controlador para fazer as tarefas necessárias para alguma requisição.

Para a criação de uma nova conta, a lógica de negócio deverá realizar as seguintes etapas:

1. Verificar se o CPF já existe. Se existir deve retornar um erro.
2. Verificar se o email já existe. Se existir deve retornar um erro.
3. Salvar as informações no banco de dados.

Para as etapas 1 e 2 utilizaremos os métodos definidos anteriormente dentro do repositório. A etapa 3 é feita utilizando o método `save()` herdado automaticamente ao estender a interface `JpaRepository`. O nosso componente ficará então da seguinte forma:

```
package talents.orange.challenge.service;

/* imports omitidos */
```

```java
@Service // declara como componente de serviço
public class UserService {

    private final UserRepository userRepository;

    @Autowired // injeção de dependência
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User create(User user) throws UniqueViolationException {
        if (userRepository.existsByCpf(user.getCpf())) {
            throw new UniqueViolationException("O CPF %s já existe no banco
        }
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new UniqueViolationException("O email %s já existe no ban
        }
        return userRepository.save(user);
    }

    public User findById(Long id) {
        return userRepository.findById(id)
            .orElseThrow(() -> new ObjectNotFoundException(id, User.class.g
    }
}
```

Da mesma forma que foi feito para o repositório, precisamos declarar a classe de serviço como um componente Spring utilizando a anotação `@Service`. Pode-se notar que a classe possui um campo privado onde será registrada uma instância do repositório feito anteriormente. Com a anotação `@Autowired` no construtor, o framework Spring automaticamente fará a injeção dessa dependência em nosso componente, assim não precisamos criar uma instância para cada componente, pois isso é gerenciado pelo framework.

O método `create(User user)` contém a lógica de negócio mencionada anteriormente. Conforme ditam as etapas, o método primeiramente verifica se o CPF e o email da nova conta já existem no banco, estourando uma exceção `UniqueViolationException` que será tratada pelo framework para retornar uma resposta com código de status e mensagem adequadas. Feito isso, basta somente persistir a nova conta no banco de dados, feito simplesmente chamando o método `save()` gerado para o repositório, sendo atribuído um valor de ID para a nova entidade. Perceba que a única validação aqui é a de CPF ou email duplicados. Veremos adiante que as validações de campos vazios, nulos, em formatos corretos, etc, serão feitas na camada do controlador, também com uma única anotação.

A exceção `UniqueViolationException` utilizada no método de registro foi criada por conveniência, para que possa ser filtrada e tratada na camada de controlador. Dessa forma será possível responder à requisição com uma mensagem de erro e código de status adequados. A sua implementação está exibida logo abaixo:

```java
package talents.orange.challenge.service.exception;

public class UniqueViolationException extends Exception {

    public UniqueViolationException(String message) {
        super(message);
    }

}
```

Além do método para salvar uma nova conta, também foi incluído o método `findById(Long id)` para buscar uma conta pelo ID no banco de dados. Assim poderemos verificar se as informações foram salvas corretamente. Esse método também estourará uma exceção `ObjectNotFoundException` caso não encontre uma conta o ID requisitado.

6. Recebendo as requisições

Com o nosso modelo, repositório e lógicas de negócio prontos, basta agora somente criar as rotas para a nossa API para que as requisições possam ser enviadas de algum site ou aplicativo mobile. Esta última etapa tratará do controlador, que além de receber as requisições, irá validar os campos conformes as anotações declaradas no nosso modelo, e passará a responsabilidade para o nosso componente de serviço. Feito isso, o controlador terá que enviar uma resposta para a requisição, dizendo se deu tudo certo ou se ocorreu algum erro.

Sem mais delongas, a seguir está parte do código para o componente de controlador que irá receber as requisições da API.

```java
package talents.orange.challenge.controller;

/* imports omitidos */

@RestController
@RequestMapping(value = "/api/user", produces = MediaType.APPLICATION_JSON_
public class UserController {

    private final UserService userService;

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

    ...

}
```

Antes de tudo, assim como foi feito para o repositório e o serviço, aqui também é preciso declara que essa classe é um componente do Spring. Para isso é utilizada a anotação `@RestController`, que, na verdade, é a união de outras duas anotações: `@Controller`, que diz para o framework que isso é um componente controlador, e `@ResponseBody` que insere o valor de retorno dos métodos no corpo da resposta.

A outra anotação utilizada aqui é `@RequestMapping`, utilizada para mapear as requisições HTTP para uma classe ou método. Nesse caso, estamos mapeando as requisições na rota `/api/user` para este componente, e também estamos declarando que o conteúdo de todas as respostas desse controlador serão do tipo JSON (em termos técnicos, o header `Content-Type` terá o valor `application/json`).

Semelhante ao que foi feito para injetar uma instância do repositório no nosso componente de serviço, aqui estaremos injetando uma instância do serviço no controlador. Esse procedimento é feito da mesma forma, por meio da anotação `@Autowired` no construtor, para que o framework realize essa injeção de dependência durante a inicialização do aplicativo.

Criando uma conta

Partindo agora para a requisição de criação de contas, precisaremos mapear uma rota para o verbo POST (pois irá criar um novo recurso), e que receberá valores no formato JSON no corpo (body) da requisição. Isso é feito através da anotação `@PostMapping`, similar à `@RequestMapping` usada anteriormente, mas que mapeia somente requisições POST. Isso se traduzirá no seguinte método abaixo, declarado dentro da classe de controlador:

```
@PostMapping(value = "/signup", consumes = MediaType.APPLICATION_JSON_VALUE
public ResponseEntity<Void> create(@Valid @RequestBody User user) throws Un
    User created = userService.create(user);
    URI uri = ServletUriComponentsBuilder.fromCurrentContextPath()
        .path("/api/user/{id}")
        .buildAndExpand(created.getId())
        .toUri();
    return ResponseEntity.created(uri).build();
}
```

Como foi dito, utilizamos a anotação `@PostMapping` para mapear uma rota POST para este método. Foi setado `value = "/signup"` para essa rota, que é incluído no final da rota da classe (ou seja: `/api/user/signup`). Além disso, também foi declarado que essa rota consumirá conteúdo somente no formato JSON com o valor `consumes`. Nos parâmetros do método são usadas mais outras duas anotações importantes: `@Valid`, que instrui o framework a fazer a validação dos campos conforme declarado anteriormente no modelo, e `@RequestBody`, que declara que o framework deve desserializar o objeto a partir do corpo da requisição (converter de JSON para POJO).

Com o objeto desserializado e validado, podemos passá-lo para o nosso serviço que será responsável por salvar no banco de dados. Após a criação da entidade, teremos um novo objeto com os mesmos valores de antes, porém, agora também com um valor de ID. A partir desse ID, podemos criar uma URI através do `ServletUriComponentsBuilder`, que direciona para o local que o novo recurso pode ser acessado.

Finalizada a lógica de negócio e a construção da URI, o controlador deve agora enviar uma resposta de volta. Utilizando o `ResponseEntity` é possível customizar de várias maneiras a resposta da sua API. Nesse caso, estaremos respondendo com um status 201 (CREATED, informa que um novo recurso foi criado), e com a URI da localização do novo recurso no cabeçalho da resposta (tecnicamente, o header `Location` conterá a URI construída).

Acessando uma conta

Podemos agora requisitar a API para criar novas contas, mas ainda não temos como recuperar os dados salvos. Para isso precisaremos mapear mais uma rota, dessa vez para o verbo GET. Isso é feito da mesma forma que foi feito para a rota de POST, mas usando agora a anotação `@GetMapping`. Nesse caso, iremos passar um ID através da URL da requisição (ex.: `/api/user/1`), que pode ser mapeado para uma variável com a anotação `@PathVariable`. No método mapeado, precisaremos apenas passar o ID para o serviço, que buscará no banco de dados com o repositório e retornará o recurso. O controlador então responderá com status 200 (OK) e com o recurso no corpo da resposta. O método responsável por isso está incluído a seguir:

```
@GetMapping("/{id}")
public ResponseEntity<User> findById(@PathVariable Long id) {
    User user = userService.findById(id);
    return ResponseEntity.ok(user);
}
```

7. Tratando os erros

Como nem tudo na vida são flores, nem todas as requisições resultarão na criação com sucesso de uma nova conta ou retornarão somente contas existentes. O último passo para a nossa a aplicação é fazer o tratamento desses casos, de forma que sejam enviadas respostas com status e mensagens de erro adequadas para quem mandou a requisição. Porém, nesses casos o framework também nos auxilia em fornecer ferramentas que facilitam essa tarefa. Com a anotação `@ControllerAdvice` podemos declarar um componente responsável por tratar as exceções que podem vir a serem estouradas durante o ciclo de vida de uma requisição.

Em nosso caso, faremos um tratamento simples para somente 2 tipos de exceção: `UniqueViolationException`, quando existe um CPF ou email duplicados, e `ObjectNotFoundException`, quando não existe um recurso com o ID requisitado. Dentro do componente, usaremos a anotação `@ExceptionHandler( /* classe da exceção /* )` para declarar algum método responsável por tratar uma ou mais classes de exceção, conforme o código abaixo:

```
package talents.orange.challenge.controller.exception;

/* imports omitidos */

@ControllerAdvice
public class ControllerExceptionHandler {
```

```
    private ResponseEntity<ApiError> handleResponse(ApiError err) {
        return ResponseEntity.status(err.getStatus()).body(err);
    }

    @ExceptionHandler(ObjectNotFoundException.class)
    public ResponseEntity<ApiError> handleObjectNotFound(ObjectNotFoundExce
        return handleResponse(ApiError.builder()
            .status(HttpStatus.NOT_FOUND.value())
            .message("Recurso %s não encontrado com ID %s".formatted(e.getE
            .build());
    }

    @ExceptionHandler(UniqueViolationException.class)
    public ResponseEntity<ApiError> handleUniqueViolation(UniqueViolationEx
        return handleResponse(ApiError.builder()
            .status(HttpStatus.BAD_REQUEST.value())
            .message(e.getLocalizedMessage())
            .build());
    }
}
```

Em suma, este componente irá receber as exceções como parâmetros dos métodos, e enviará uma resposta à requisição com o status e mensagem adequados. No caso de recurso não encontrado, a resposta terá status 404 (NOT FOUND). E no caso de CPF ou email duplicados, a resposta terá status 400 (BAD REQUEST), com a mensagem já inclusa na exceção, definida no código do serviço.

A classe `ApiError` é um objeto simples utilizado para definir os campos que a resposta de erro terá. Seu código segue logo abaixo:

```
package talents.orange.challenge.controller.exception;

/* imports omitidos */

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ApiError implements Serializable {

    @Serial
    private static final long serialVersionUID = 1L;
    private final LocalDateTime timeStamp = LocalDateTime.now();
    private Integer status;
```

```
    private String message;
}
```

8. Rodando a aplicação

Agora que temos o código da aplicação pronto, basta somente colocarmos ela para funcionar e testar para verificar se está tudo conforme o esperado. Para isso, abra o terminal do seu sistema, navegue até a pasta do projeto e execute o comando do wrapper do Gradle (`gradlew`) para iniciar a aplicação.

```
# Navegar até a pasta do projeto
cd <caminho até a pasta do projeto>
# No Linux ou no Powershell
./gradlew bootRun
# Ou no Windows, pelo Prompt de Comando
gradlew.bat bootRun
```

Após alguns instantes a aplicação deve estar iniciada. Agora podemos enviar algumas requisições para ver se tudo está funcionado. Podemos fazer isso através de aplicações gráficas como o Postman ou Insomnia, ou no caso desse artigo, utilizar o cURL na linha de comando.

Vamos agora enviar a primeira requisição, criando uma nova conta com valores corretos para passar pela validação. Esperamos que a aplicação retorne com status 201 e com um link para acessar o recurso criado. Para gerar um CPF válido existem sites que permitem fazer isso, nesse caso estarei usando o site https://www.geradordecpf.org/. Como estamos rodando localmente, as requisições devem ser enviadas para a URL `http://localhost:8080/`.

```
curl --request POST \
  --url 'http://localhost:8080/api/user/signup' \
  --header 'Content-Type: application/json' \
  --data '{
    "name": "Fulano Ciclano",
    "cpf": "34836677254",
    "email": "fulano@email.com",
    "birthday": "1990-01-01"
  }'
================================================
< HTTP/1.1 201
< Location: http://localhost:8080/api/user/1
```

Como podemos ver, obtemos a resposta esperada. O servidor recebeu a requisição e registrou a nova conta, retornando o link através do header `Location`. Fazendo agora outra requisição para o link informado, veremos que as informações estão corretas no banco de dados:

```
curl --request GET --url http://localhost:8080/api/user/1
=================================================================
< HTTP/1.1 200
< Content-Type: application/json
{
  "name": "Fulano Ciclano",
  "email": "fulano@email.com",
  "cpf": "34836677254",
  "birthday": "1990-01-01"
}
```

Ótimo, a conta foi criada e as informações foram salvas corretamente. Vamos ver agora o que acontece se tentarmos cadastrar novamente a mesma conta:

```
< HTTP/1.1 400
< Content-Type: application/json
{
  "timeStamp": "2021-01-06T18:25:27.6047912",
  "status": 400,
  "message": "O CPF 34836677254 já existe no banco de dados."
}
```

Diferentemente da primeira vez, agora o servidor retornou um erro, com a mensagem de CPF duplicado. Se mudarmos somente o CPF, receberemos uma outra mensagem, dessa vez por causa do email duplicado:

```
< HTTP/1.1 400
< Content-Type: application/json
{
  "timeStamp": "2021-01-06T18:28:09.6897737",
  "status": 400,
  "message": "O email fulano@email.com já existe no banco de dados."
}
```

Agora para um último caso, vamos enviar uma requisição com todos os campos inválidos e ver o que acontece:

```
curl --request POST \
  --url http://172.31.112.1:8080/api/user/signup \
  --header 'Content-Type: application/json' \
  --data '{
    "name": "",
```

```
    "cpf": "12345678910",
    "email": "fulano.email.com",
    "birthday": "2077-01-01"
}'
================================================
< HTTP/1.1 400
< Content-Type: application/json
{
  "timestamp": "2021-01-06T21:32:10.780+00:00",
  "status": 400,
  "error": "Bad Request",
  "message": "Validation failed for object='user'. Error count: 4",
  "errors": [
    {
      ...
      "defaultMessage": "deve ser um endereço de e-mail bem formado",
      "field": "email",
      ...
    },
    {
      ...
      "defaultMessage": "número do registro de contribuinte individual bras
      "field": "cpf",
      ...
    },
    {
      ...
      "defaultMessage": "deve ser uma data passada",
      "field": "birthday",
      ...
    },
    {
      ...
      "defaultMessage": "não deve estar em branco",
      "field": "name",
      ...
    }
  ],
  "path": "/api/user/signup"
}
```

Como a resposta para esse caso é um pouco extensa, alguns dos valores foram omitidos aqui. Porém, podemos ver mesmo assim que o servidor nos fornece bastante informação do que

causou o erro. Cada campo inválido possui uma mensagem descrevendo o que deve ser feito para corrigir. No caso do CPF, mesmo possuindo o número de dígitos necessários (11), ainda assim o validador consegue identificá-lo como inválido.

9. Concluindo

Após todos esses testes, podemos ver que a aplicação funciona e responde corretamente às requisições. Com a ajuda do framework Spring, foi possível desenvolver uma API REST robusta, rapidamente e com pouco esforço. Mesmo com uma base simples como essa, ainda há muita possibilidade para incrementar e expandir. Poderíamos, por exemplo, utilizar um outro módulo como o Spring Security, para adicionar um sistema de autenticação e autorização para nossa API, podendo integrar uma autenticação por JWT, OAuth ou Single Sign-On.

Todo o código presente neste artigo também está disponível no meu Github, através do link https://github.com/ggwadera/zup-desafio-orange-talents. Nesse repositório, além do código para a API, também estará disponível os testes unitários usados para testar a aplicação durante o desenvolvimento, mas que não foram citados aqui.

A API também está disponível online na plataforma Heroku, onde você pode testar sem precisar rodar nada localmente. Para isso, utilize o link https://zup-desafio.herokuapp.com/ (para criar uma conta, por exemplo, utilize https://zup-desafio.herokuapp.com/api/user/signup).

Adicionalmente, também é disponibilizado uma imagem Docker para rodar a aplicação. Para isso utilize o comando abaixo.

```
docker run --rm -it -p 8080:8080 ggwadera/zup
```