Autor: Gabriel Estêvão de Morais Rodrigues

Link Projeto: https://github.com/GabrielEstmr/sistemaVacina

1. **Contexto:**

Criar serviço de API REST para controlar a aplicação de vacinas na população

2. **Requisitos:**

- Registro dos usuários e de vacinas devem ser feitos, onde:
- Para Usuários, tem-se que: precisam ser registradas nome, e-mail, CPF e data de nascimento, onde e-mail e CPF devem ser únicos.
- Para Vacinas, tem-se que: obrigatórios dados de nome da vacina, e-mail do usuário e a data que foi realizada a vacina.
- Dois endpoints devem ser disponibilizados, um para cada tabela.

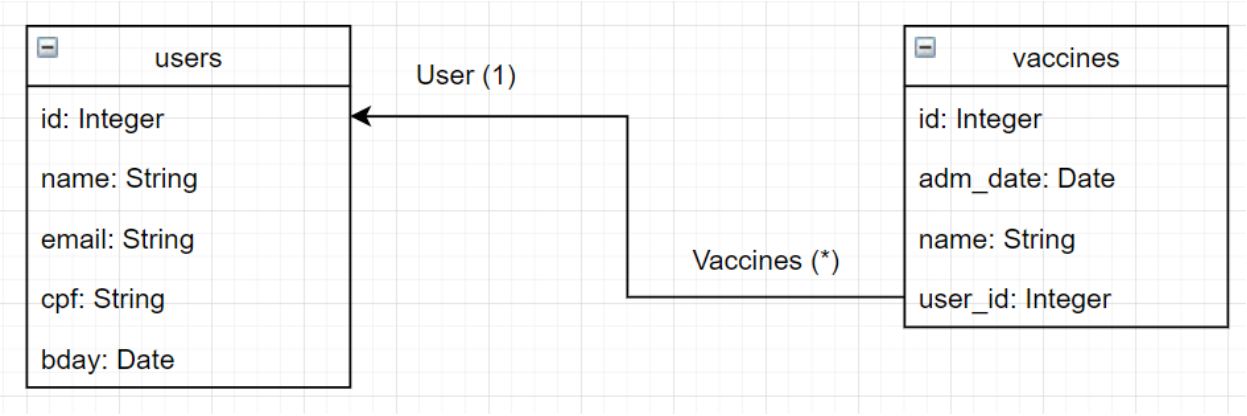
3. **Considerações:**

O usuário pode cadastrar mais de um registro de vacina (para o caso de haver necessidade de tomar mais de uma vacina diferente ou mais de uma dose da mesma vacina);

A data de administração da vacina não é necessariamente a data de registro da dose no banco de dados, para abrir a possibilidade de registros retroativos.

4. **Modelo do Banco de Dados:**

Foi usado um Banco de dados MySQL, relacional. Com base nos requisitos e considerações, tem-se o seguinte modelo:



Onde:

- bday, na tabela users: se relaciona à data de aniversário do usuário;
- adm_date, na tabela vacines: se relaciona à data de administração da vacina pelo usuário;

5. **Endpoints:**

Seguindo os requisitos, foram disponibilizados os endpoints para cadastro dos dados. De forma complementar foram disponibilizados os métodos para atualizar, deletar, ler e listar os dados de ambas as tabelas. Sendo assim, tem-se:

Para Vacinas (Tabela Vaccines):

- http://localhost:8080/vaccines, método POST/Create: Criação de um novo registro de Vacina no banco da dados, com o seguinte body no formato JSON:

  {

  "name": "ac",

  "admDate": "2021-02-27T17:11:33.594+00:00",

  "userEmail": "gabriel.estmr@gmail.com"

  }

- http://localhost:8080/vaccines, método PUT/Update: Atualização de um registro de Vacina no banco da dados, com o seguinte body no formato JSON:

  {

  "name": "ac",

  "admDate": "2021-02-27T17:11:33.594+00:00",

  "userEmail": "gabriel.estmr@gmail.com"

  }

- http://localhost:8080/vaccines/{id} , método DELETE/Delete: Deleção de um registro de Vacina no banco da dados;
- http://localhost:8080/vaccines/{id} , método GET/Read: Leitura de um registro de Vacina no banco da dados;
- http://localhost:8080/vaccines/, método GET/Read: Listagem de todos os registros de Vacina no banco da dados; (nesse caso não foi feita paginação para não complicar o server);
-

Para Usuários (Tabela Users):

- http://localhost:8080/users, método POST/Create: Criação de um novo registro de Usuário no banco da dados, com o seguinte body no formato JSON:

  {

"name": "ZUP Employee",

"email": "zup@zup.com",

"cpf": "100.110.999-90",

"bday": "2021-02-27T17:11:33.594+00:00"

}

- http://localhost:8080/users, método PUT/Update: Atualização de um registro de Usuário no banco da dados, com o seguinte body no formato JSON:

{

"name": "ZUP Employee Novo Nome",

"email": "zup@zup.com",

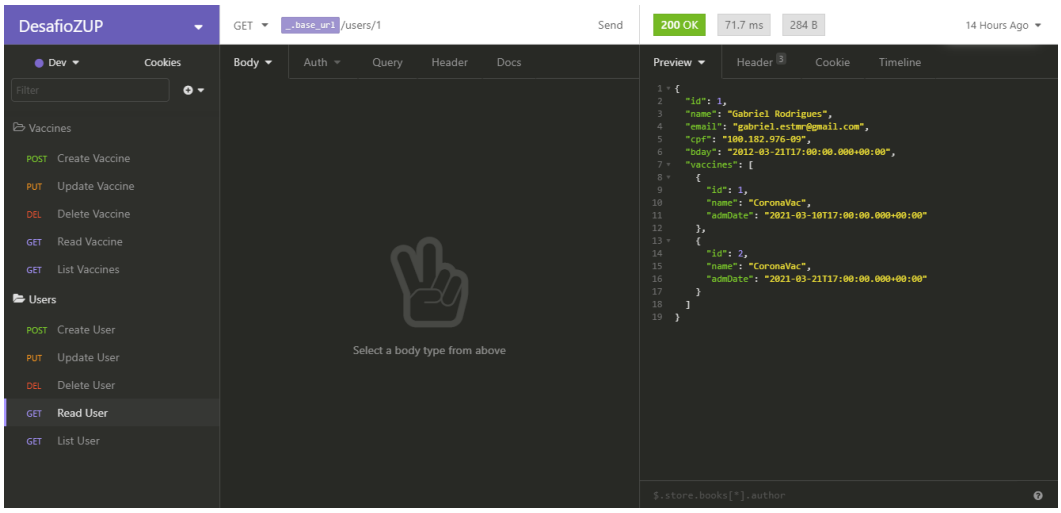
"cpf": "100.100.999-09",

"bday": "2021-02-27T17:11:33.594+00:00"

}

- http://localhost:8080/users/{id} , método DELETE/Delete: Deleção de um registro de Usuário no banco da dados;
- http://localhost:8080/users/{id} , método GET/Read: Leitura de um registro de Usuário no banco da dados;
- http://localhost:8080/users/, método GET/Read: Listagem de todos os registros de Usuários no banco da dados; (nesse caso não foi feita paginação para não complicar o server);
-

Esquematização dos endpoints no software Insomnia:



6. **Tecnologias Usadas:**

Spring Web < org.springframework.web>: Permite o Mapeamento dos métodos REST dentro de uma classe atribuída como controller no JAVA, isto é, permite a ligação entre um endpoint e um método de um Service que é chamado através dos controllers. A vantagem do Spring Web é a facilidade de configuração dos parâmetros (Body, QueryParams, etc...), sendo feita de forma facilitada através da chamada dos decorators do pacote; Seus principais métodos são:

- @RestController: atribui uma classe do JAVA à função controller
- @RequestMapping: determinação do endpoint a seu usado pelo controller, seu queryparams método REST

Bean Validation < javax.validation>: Biblioteca que permite a validação de campos de uma maneira mais direta(modalidade de validação sem acesso a dados). Na arquitetura utilizada, a validação de campo sem acesso a dados foi feita na camada de Controller, sendo aplicada dentro dos DTOS; A biblioteca contém os principais métodos de validação utilizados, o que otimiza o tempo de programação; É utilizado nos DTOs da aplicação;

JPA < org.springframework.data.jpa>: É uma espécie de ORM (Object-relational mapping) dentro da linguagem JAVA. Fornece métodos para o fluxo de dados entre o server e o banco. É utilizado mais nas camadas de repository (para uso dos métodos de acesso a dados) ;

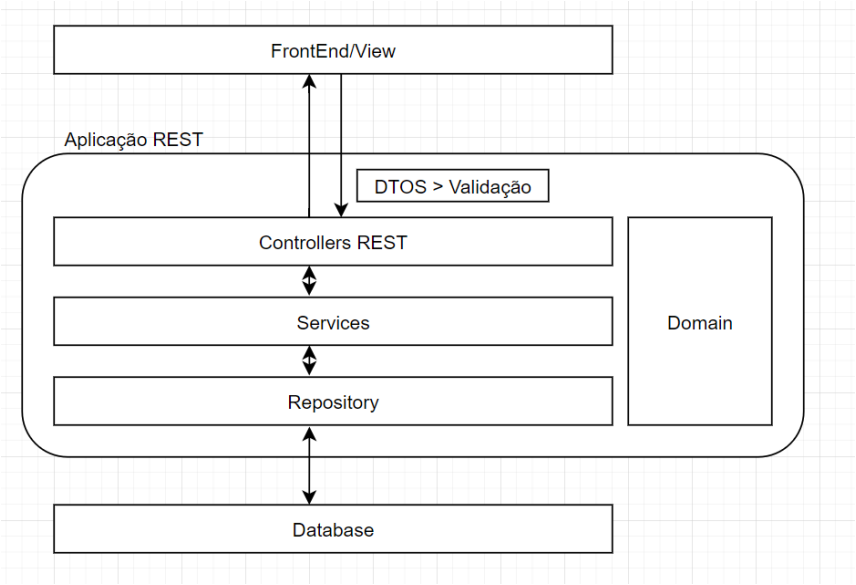
Pacote javax.persistence: permitir a criação de tabelas dentro do banco de dados a partir de objetos dentro da programação JAVA; Usado nas classes de repositor para instanciação das classes de domínio da aplicação (para criar das tabelas no banco de dados a partir de tais classes);

MAVEN: Este gerenciador de bibliotecas externas permite a instalação automática de bibliotecas de terceiros através do arquivo POM.XML. Com isso podemos adicionar mais funcionalidades no nosso código e otimizar o tempo de desenvolvimento.

Banco de Dados H2: Banco de Dados baseado no MySQL, alocado na memória; Sua principal vantagem é a fácil manipulação para o ambiente de desenvolvimento;

7. **Estrutura da Aplicação:**

Seguindo as boas práticas de programação de desenvolvimento de softwares, a estrutura utilizada para este server foi a seguinte:



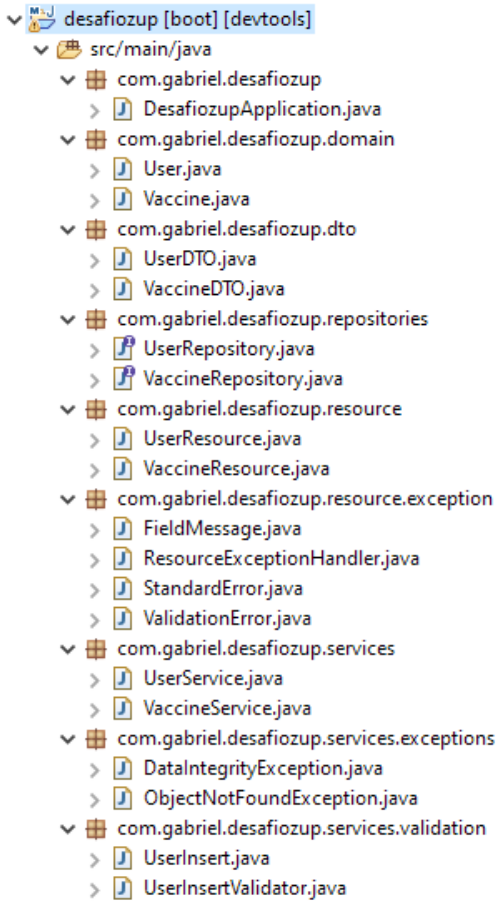
Onde:

- **DTO**: Data-Tranfer-Objects: São classes de objetos que nos permite o fluxo de dados no formato mais pertinente para a aplicação. É onde a validação da modalidade sem acesso aos dados é feita através da biblioteca de BeanValidation;
- **Controllers**: São os controladores da aplicação, ou seja, onde há a ligação entre os endpoints e a chamada dos métodos contidos nos Services;
- **Services**: São os serviços da aplicação, ou seja, onde está contido a regra de negócio e validação de acesso aos dados; A lógica da regra de negócio é implementada através dos métodos contidos nos repositórios/repositor. Não possui contato com tecnologia nenhuma (infra), como o JPA;
- **Repositories**: Os repositórios contêm os métodos de acesso a dados. Normalmente são uma mistura de métodos já pré-existentes no JPA do Spring (FindById, Save, etc) e métodos personalizados da aplicação;
- **Camada de Domínio**: Instanciação das classes de domínio e geração do banco de dados via pacote javax.persistence;

Observação: Tratativa de erros:

- Validação: via pacote Bean Validation: Controllers REST (via DTO);
- Validações customizadas: Via service;

Além disso foi usado o Design-Pattern baseado nos domínios da aplicação (DDD – Domain Driven Design). Isso se traduz da divisão da estrutura acima apresentada em domínios, neste caso os domínios de Usuários e Vacina, conforme apresentado na árvore de arquivos abaixo:



8. **Implementação das Classes:**

Para praticamente todas as classes seguiu-se o seguinte procedimento: (Com exceção dos controller e da classe principal):

- Criação de Construtor vazio, pois há bibliotecas que o utilizam;
- Criação de construtor a partir de seus campos, para instanciação da classe dentro de outra classe;
- Criação dos Getters e Setters, para manipulação dos atributos de cada classe
- Implementação do Serializable: para classe ser convertida em sequência de bites, ou seja, ser gravados em arquivos e trafegar em redes;
- Criação do HashCode and Equals (Apenas para Ids): permite comparação de um objeto por seu conteúdo e não pelo ponteiro de memória. Implementado apenas para o Id pois é o campo de comparação de cada classe.

8.1. Classes Domínio:

A notação `@Entity` do pacote javax.persistence faz a ligação entre a classe de domínio e a instanciação da tabela dentro do banco de dados, segundo o trecho de código abaixo:

```
@Entity
public class User implements Serializable{
```

Para a instanciação do atributo da classe ao id da tabela, tem-se o seguinte código:

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

Neste Caso, o método `@GeneratedValue` gera automaticamente um valor para o campo id, sem que seja necessário sua atribuição. Para isso foi utilizado o método Identity, suportado pelo banco H2 (estrutura MySQL);

Outra funcionalidade muito importante do pacote javax.persistence é a atribuição de valores únicos nas colunas" `@Column(unique = true)`", o que foi feito para email e bday, seguindo os requisitos.

```
@Column(unique = true)
private String email;

@Column(unique = true)
private String cpf;
private Date bday;
```

Outra funcionalidade do javax.persistence é o controle de associação de colunas entre as tabelas. No caso do problema proposto, tem-se um relação "um para muitos" em relação a tabela de usuários. A notação `@JsonManagedReference` evita a renderização cíclica dessa associação.

```
@JsonManagedReference
@OneToMany(mappedBy = "user")
private List<Vaccine> vaccines = new ArrayList<>();
```

Para a classe de Vacinas, tem-se:

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

8.2. Classes DTOs:

A classe DTOs foi utilizada por dois motivos: validação sem acesso a dados dos campos através da biblioteca BeanValidation < javax.validation> e formatação do retorno da API (no caso da listagem de Usuários, que não queríamos com as vacinas associadas);

Validação Para o Usuário:

```
@NotEmpty(message = "Campo deve ser preenchido.")
@Length(min = 5, max = 80, message = "Campo deve conter entre 5 e 80
caracteres.")
```

```
	private String name;

	@NotEmpty(message = "Campo deve ser preenchido.")
		@Length(min = 7, max = 80, message = "Campo deve conter entre 7 e 80
caracteres.")
	@Email(message = "Um email valido deve ser preenchido.")
	private String email;

	@Length(min = 14, max = 14, message = "CPF deve ser no formato XXX.XXX.XXX-
XX")
	@NotEmpty(message = "Campo deve ser preenchido.")
	private String cpf;

	@NotNull(message = "Campo deve ser preenchido.")
	@Temporal(TemporalType.TIMESTAMP)
	private Date bday;
```

Validação para os campos de Vacina:

```
	private Integer id;

	@NotEmpty(message = "Campo deve ser preenchido.")
	@Length(min = 5, max = 80, message = "Campo deve conter entre 5 e 80
caracteres.")
	private String name;

	@NotNull(message = "Campo deve ser preenchido.")
	@Temporal(TemporalType.TIMESTAMP)
	private Date admDate;

	@NotEmpty(message = "Campo deve ser preenchido.")
	@Length(min = 7, max = 80, message = "Campo deve conter entre 7 e 80
caracteres.")
	@Email(message = "Um email valido deve ser preenchido.")
	private String userEmail;
```

8.3. Classes Repository

Nas classes Repository, foi usado o decorator `@Repository` para a instanciação desta classe como um repositório do Spring para que seja possível a posterior injeção de dependência em outras classes. Além disso, para o repositório de usuário, foi necessário criar dois métodos personalizados: “findByEmail” e “findByCpf” para a posterior validação de e-mail e CPF repetidos no service;

```
	package com.gabriel.desafiozup.repositories;

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

	import com.gabriel.desafiozup.domain.User;

	@Repository
	public interface UserRepository extends JpaRepository<User, Integer>{

		//Transactional faz transação ficar mais rápida e diminui loking do BD
```

```
	@org.springframework.transaction.annotation.Transactional(readOnly =
true)
	User findByEmail(String email);


	@org.springframework.transaction.annotation.Transactional(readOnly =
true)
	User findByCpf(String cpf);

}
```

Além disso, o método `extends JpaRepository` permite a soma dos métodos já contidos no JPA e os métodos personalizados no repositório, como evidenciado abaixo:

```
public interface UserRepository extends JpaRepository<User, Integer>
```

8.4. Classes Service

Nas classes Services, foi usado o decorator `@Service` para a instanciação desta classe como um repositório do Spring, como apresentado abaixo:

```
@Service
public class UserService {
```

Também é usado a notação `@Autowired` para a injeção de dependências para injetar o repositório do respectivo domínio para que possamos usar seus métodos, conforme a notação abaixo:

```
@Autowired
	private UserRepository repo;
```

Assim podemos usar os métodos do repositório para criar as regras de negócio e validações com acesso a dados no servisse. No caso do service de Usuários, foram feitos os métodos “find”, “insert” “update” “delete” e “List” onde, no método “insert” foram colocadas as validações que tratam a inserção de CPF e E-mail repetidos pelo usuário, usando os métodos personalizados do repositor de usuários (UserRepository)

8.5. Classes Resource (Controllers)

No caso das classes controllers, tem-se a notação `@RestController` para a identificação da classe como controller pelo Spring Web e a função `@RequestMapping(value = "/users")` para a determinação do endpoint que será adotado por essa classe.

```
@RestController
@RequestMapping(value = "/users")
public class UserResource {
```

Mais um vez tem-se a injeção de dependência, desse vez para injetar os métodos do service no controller, segundo a notação notação `@Autowired`:

```
@Autowired
	private UserService service;
```

Dentro da classe UserResource, foram criados outros métodos públicos, onde o `@RequestMapping` determina os outros parâmetros do endpoint, tais como: Método, QueryParams, Body, entre outros. Levando em consideração o método POST do domínio de usuários, tem-se que:

```
@RequestMapping(method = RequestMethod.POST)
	public ResponseEntity<Void> insert(@Valid @RequestBody UserDTO
objDTO){
		User obj = service.fromDTO(objDTO);
		obj = service.insert(obj);
		URI uri = ServletUriComponentsBuilder.fromCurrentRequest()

	.path("/{id}").buildAndExpand(obj.getId()).toUri();

		return ResponseEntity.created(uri).build();
	}
```

Onde:

- `@Valid` valida a entrada via DTO, segundo o Bean Validation utilizado na classe de DTO
- `@RequestBody` determina que o body da requisição será do tipo UserDTO
- `fromDTO:` método do service que transforma o UserDTO em um objeto da classe User
- `service.insert(obj):` método do service de usuários que contém as validações necessárias de CPF e E-mail;
- `URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(obj.getId()).toUri():` retorno do id criado pelo método;
- `ResponseEntity:` tipo especial do Spring que já armazena/encapsula várias informações do tipo REST: resposta, erros etc.;

8.6. Implementação de exceção padronizada:

Para a padronização do formato da resposta do server, foram feitas classes de padronização das exceções, tanto para as exceções do controller quanto para as exceções do service:

```
public class ObjectNotFoundException extends RuntimeException{

	private static final long serialVersionUID = 1L;

	public ObjectNotFoundException(String msg) {
		super(msg);
	}

	public ObjectNotFoundException(String msg, Throwable cause) {
		super(msg,cause);
	}
}
```

- `extends RuntimeException`: estrutura e exceções da linguagem JAVA: extende classe de erros RuntimeException default
- `@ExceptionHandler(ObjectNotFoundException.class)`: dizer que é um tratador de exceções do tipo de classe `ObjectNotFoundException;`

8.7. Outras observações importantes:

- Endpoints: foram colocados no plural por ser padrão do mercado;
- As seguintes dependências foram colocadas no arquivo pom.XML para carregamentos das bibliotecas pelo Maven:

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-test</artifactId>
	<scope>test</scope>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
	<scope>runtime</scope>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-devtools</artifactId>
</dependency>

<dependency>
	<groupId>org.hibernate.validator</groupId>
	<artifactId>hibernate-validator</artifactId>
	<version>6.1.2.Final</version>
</dependency>
```

- `public class DesafiozupApplication implements CommandLineRunner{`: permite rodar um código quando a aplicação principal iniciar para criação de alguns dados de teste (pois o banco de dados H2 é alocado na memória;
- Várias vacinas para apenas um usuário (user_id: nome da chave estrangeira):

```
@ManyToOne
@JoinColumn(name = "user_id")
```

- `@OneToMany(mappedBy = "user")` : nome do atributo da classe estado que gerou a chave estrangeira de assossiação

- Proteção contra a resialização ciclica: Users pode serializar as vacinas relativas a ele, mas as Vaccines nao pode serializar os Users relativos a ela:

  `@JsonManagedReference > libera serialização`

  `@JsonBackReference > trava serialização`