Criando uma API que controla aplicação de vacinas entre a população brasileilra.

Nesse artigo quero propor um guia básico, porém detalhado, para a construção de uma API (Application Programming Interface) arquitetada em REST (Representational State Transfer). Sempre que possível vou incluir referências tanto de documentação quanto de blogs que achei interessantes e que me ajudaram a construir meu projeto.

Um adendo: se não todas, a maioria dos materiais que usei de referência estão em língua inglesa. Meus, caros, aprender inglês é inescapável, portanto se acostume com o fato de que você vai ter de ler muito em inglês no seu dia a dia. E por isso mesmo decidi escrever também em inglês no meu código, o que na minha opinião já um bom jeito de se acostumar com a língua de Shakespeare.

Antes de começar, quero deixar claro que você pode acessar todo o código fonte de minha API no meu GitHub. Lá você vai ver como foi que organizei meu GitFlow e como foi que tentei colocar em prática o conceito de Versionamento Semantico. Por favor dê uma olhada lá!

Dito tudo isso é hora de começar. E a primeiríssima coisa a se fazer é abrir acessar https://start.spring.io/. Assumindo que você já sabe como o Spring Initializr funciona (caso não, de uma olhada aqui), partimos para o que interessa.

Para nossa aplicação vamos usar o Apache-Maven como gerenciador de dependências; Java na versão 11 como nossa linguagem; Spring Boot versão 2.4.3 (última versão estável na data que em que escrevo); Jar como packaging. Fica a seu critério a escolha de nomes de grupos e artefatos.

Finalmente, às dependências: Spring Web, Spring Data Jpa, Validations e H2 Database.

Depois que você clicar em gerar e extrair seus arquivos, seu pom.xml deverá ter mais ou menos essa cara:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
  </properties>
```

```
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Explicando o porquê das dependências:

Spring Web: com esta dependência é possível criar aplicações compatíveis com a web, ou seja, ela trás consigo uma variedade de ferramentas relacionas como o desenvolvimento web. Spring Web faz uso do Spring MVC, da arquitetura REST e do Tomcat como servidor padrão. Aqui você encontra um breve exemplo explicativo: https://spring.io/guides/gs/rest-service/.

Spring Data Jpa: o uso dessa dependência facilita a implementação de repositórios baseados na JPA (Java Persistency Application Programming Interface). Com ela é possível fazer operações com seu banco de dados. Aqui você encontra um breve exemplo explicativo: https://spring.io/guides/gs/accessing-data-rest/.

Validations: com esta dependência é possível que você use alguns “filtros” em sua aplicação, alcançando resultados de maior precisão. Aqui você encontra um breve exemplo explicativo: https://spring.io/guides/gs/validating-form-input/.

H2 Database: é uma tipo de banco dados que funciona em memória, ela é perfeita para ocasiões de treinos, como é o exemplo desse artigo. Aqui você encontra um breve exemplo explicativo: https://www.baeldung.com/spring-boot-h2-database. Mas calma, pequeno gafanhoto, vamos usar esse banco de dados de forma provisória. Mais pra frente vou lhe mostrar como configurar aplicação para usar um outro banco de dados, o MySql. O ponto positivo do H2 é que por padrão ele funciona “out of the box” ou com pouquíssima configuração.

E essa é a base para a construção de nossa aplicação. É hora de partir para o código.

Como vai funcionar: da mesma forma como já aconteceu aqui com o arquivo pom.xml, primeiro mostro o arquivo ou código e depois o explico em detalhes. Essa será a minha estratégia daqui pra frente. Tentei não fazer muitos comentários no código em si para evitar poluição visual. Mesmo assim a formatação dele não é a ideal. Peço sua paciência e compreensão na leitura arquivos. Obrigado!
Construindo as entidades:

`src/main/java/io/orangehealth/bva/domain/User.java`

```
/**
 * User Entity
 *
 * @author Rafael Rodrigues
 */

@Entity
@Table(name = "USER")
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "user_id")
	private long id;

	@Column(name = "first_name")
	@NotBlank(message = "First name is a required field.")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Column(name = "email", unique = true)
	@NotBlank(message = "E-mail is a required field.")
	@Email(message = "Invalid e-mail")
	private String email;

	@Column(name = "cpf", unique = true)
	@NotBlank(message = "CPF is a required field")
	@Pattern(regexp = "^\\d{3}\\.\\d{3}\\.\\d{3}\\-\\d{2}$",
			message = "000.000.000-00 must be the specified format for CPF.")
	private String cpf;

	@Column(name = "birth_date")
	@NotNull(message = "Birth date is a required field.")
	@DateTimeFormat(pattern = "yyyy-MM-dd")
	private LocalDate birthDate;

	@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
	@JsonIgnoreProperties("user")
	private List<VaccineInjection> vaccines = new ArrayList<>();

	/**
	 * Default Constructor
	 */
	protected User() {};

	/**
	 * Constructor without VaccineInjection object
	 * @param firstName
	 * @param lastName
	 * @param email
	 * @param cpf
	 * @param birthDate
	 */
```

```
	public User( String firstName, String lastName, String email, String cpf,
			LocalDate birthDate) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
		this.cpf = cpf;
		this.birthDate = birthDate;
	}

	/**
	 * Complete Constructor
	 *
	 * @param firstName
	 * @param lastName
	 * @param email
	 * @param cpf
	 * @param birthDate
	 * @param vaccine
	 */
	public User(String firstName, String lastName, String email, String cpf,
			LocalDate birthDate, VaccineInjection vaccine) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
		this.cpf = cpf;
		this.birthDate = birthDate;
		this.vaccines = Arrays.asList(vaccine);
	}

	public void addVaccine(VaccineInjection vaccine) {
		vaccines.add(vaccine);
		vaccine.setUser(this);
	}

	public void removeVaccine(VaccineInjection vaccine) {
		vaccines.remove(vaccine);
		vaccine.setUser(null);
	}


	// Getters and Setter
}
```

Vamos lá, parece muito, mas, na verdade, nem é tanto assim. A primeira coisa é nome da tabela discriminado: “USER”. Logo em seguida, nessa entidade há 7 campos, sendo: id, primeiro nome, último nome, e-mail, cpf, data de nascimento e uma lista de vacinas. Cada um desses campos carregam consigo algumas anotações.

Em id, identificamos que esse campo é de fato um id e que o valor atribuído a este campo é gerado automaticamente. O nome da coluna é “user_id”.

Em primeiro nome, temos o nome da coluna, “last_name”, e uma condição de que este campo não pode ser nulo e de conter pelo menos um caractere que não seja um espaço com a anotação @NotBlank. Caso o campo não seja preenchido criei uma mensagem de erro.

Em último nome, temos apenas o nome da coluna na tabela, “last_name”. Este campo não é um campo obrigatório, sendo facultativo a inserção dele pelo usuário. Mesmo a assim a organização pode recomendar o seu preenchimento no View da aplicação.

Em e-mail, temos o nome da coluna “e-mail”, também especificamos que não pode ser um campo nulo e deve conter pelo menos um caractere que não seja um espaço e assim como no primeiro nome também foi criada uma mensagem de erro caso não haja a inserção desse campo.
Em CPF, temos um caso muito parecido com o do e-mail, porém aqui também especificamos um padrão em expressão regular para o formato do CPF. Basicamente o que aquela regex quer dizer é: “para um começo de uma string, encontre qualquer digito numérico que se repita 3 vezes, logo depois encontre um ponto, seguido novamente de qualquer digito numérico que se repita 3 vezes, seguido de um ponto, e mais uma vez qualquer digito numérico seguido de um ponto, um travessão e finalmente qualquer digito numérico que se repita duas vezes no fim de uma string” (Ufa!). Temos uma mensagem de erro também que explica qual é formato aceitável para esse campo.

Em data de nascimento temos o nome da coluna, mensagem de erro e também um formato para o objeto LocalDate. Note que não usei um objeto do tipo String.

O último campo, vacinas, é uma lista de objetos de Aplicação de vacinas. Aqui temos algumas anotações interessantes: @OneToMany e @JsonIgnoreProperties. Em bancos de dados relacionais as tabelas, como já se intui pelo nome do banco de dados, têm relações umas com as outras, como “muitos para muitos” “um para muitos”, e @OneToMany cria exatamente esta última, “um para muitos”. O que isso quer dizer aqui no contexto de nossa aplicação é que esta entidade é uma para muitas entidades de aplicação de vacinas, entidade que você já verá. Basta você inserir esta anotação e deixar a API de persistência fazer o trabalho por você. A anotação @JsonIgnoreProperties("user") previne que entremos em um loop no carregamento da entidade, o que causa um erro.

Em seguida temos o construtor padrão deste objeto. Note que ele é protegido. Aqui meu intuito foi não correr o risco de que haja criações de objetos do tipo User vazios. Para isso criei dois outros construtores e estes devem ser usados por outras classes para criação de usuários.

No segundo construtor, temos todos os campos com exceção do campo de vacinas. Assim é possível criar um usuário que ainda não tomou nenhuma vacina. No terceiro construtor vemos que ele é completo, tem todas os campos. Com ele é possível cadastrar um usuário que já tomou alguma vacina.

Os dois últimos métodos que mostrei no arquivo são os métodos para adicionar e remover uma vacina de um usuário. Porquê criei a possibilidade de criação de um usuário que ainda não tomou uma vacina, é muito importante que também criemos estes métodos para atualizá-los no futuro.

Depois disso basta criar os Getter e Setter dos campos criados. Você vai entender melhor o que está em jogo nessa entidade quando formos criar nossos serviços e controladores.

`/src/main/java/io/orangehealth/bvac/domain/VaccineInjection.java`

```
/**
 * Vaccine Injection Entity
 *
 * @author Rafael Rodrigues
 */

@Entity
@Table(name = "VACCINE_INJECTION")
public class VaccineInjection {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "vaccine_id")
```

```
	private long id;

	@Column(name = "vaccine_name")
	@NotNull(message = "Vaccine name is a required field.")
	private Vaccine vaccineName;

	@Column(name = "injection_date")
	@NotNull(message = "Injection date is a required field.")
	@DateTimeFormat(pattern = "yyyy-MM-dd")
	private LocalDate injectionDate;

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "user_id")
	@JsonIgnoreProperties("vaccines")
	private User user;

	/**
	 * Default Constructor
	 */
	protected VaccineInjection() {};

	/**
	 * Constructor without User object
	 *
	 * @param vaccineName
	 * @param injectionDate
	 */
	public VaccineInjection(Vaccine vaccineName, LocalDate injectionDate) {
		super();
		this.vaccineName = vaccineName;
		this.injectionDate = injectionDate;
	}

	/**
	 * Complete Constructor
	 *
	 * @param vaccineName
	 * @param administrationDate
	 * @param user
	 */
	public VaccineInjection(Vaccine vaccineName, LocalDate administrationDate,
			User user) {
		super();
		this.vaccineName = vaccineName;
		this.injectionDate = administrationDate;
		this.user = user;
	}

	/* TODO: Equality Consistency
	 * reference: https://web.archive.org/web/20170524072455/http://www.onjava.com/pub/a/
onjava/2006/09/13/dont-let-hibernate-steal-your-identity.html?page=1
	 *
	 * @Override
	 * public boolean equals(Object obj) {
	 *     // TODO Auto-generated method stub
	 *     return super.equals(obj);
	 * }
	 *
	 * @Override
	 * public int hashCode() {
	 *     // TODO Auto-generated method stub
	 *     return super.hashCode();
	 * }
	 */

	// Getters and Setters
```

```
}
```

Aqui, o caminho tomado para a construção da entidade não foge em nada daquele usado para a criação da entidade de usuários. Temos 4 campos: id, nome da vacina, data de aplicação e usuário.

Em id, seguimos exatamente os passos de criação do id em User, como expliquei logo acima.

Em nome de vacina temos algo interessante mais uma vez. A lista de nomes de vacinas é uma enumeração. Isso porque não queremos que o usuário da aplicação insira qualquer coisa nesse campo e é aconselhável na camada View que as opções sejam uma espécie de botão dropdown para que não haja erros. Dê uma olhada neste Enum:

`/src/main/java/io/orangehealth/bvac/domain/Vaccine.java`

```
/**
 * Enumeration of Vaccines
 *
 * @author Rafael Rodrigues
 */

public enum Vaccine {
	Pfizer_BioNTech("Pfizer–BioNTech"), Moderna("Moderna"),
	Oxford_AstraZeneca("Oxford–AstraZeneca"),
	Sputnik_V("Sputnik V"), Johnson_n_Johnson("Johnson & Johnson"),
	Convidicea("Convidicea"), BBIBP_CorV("BBIBP-CorV"),
	CoronaVac("CoronaVac"), Covaxin("Covaxin"), CoviVac("CoviVac"),
	EpiVacCorona("EpiVacCorona"),
	RBD_Dimer("RBD-Dimer"), ;

	private String name;

	Vaccine(String name) {
		this.name = name;
	}

	public static Vaccine findByVaccineName(String vaccineName) {
		for (Vaccine vaccine : Vaccine.values()) {
			if (vaccine.name.equalsIgnoreCase(vaccineName))
				return vaccine;
		}
		return null;
	}

	public String getName() {
		return name;
	}
}
```

Aqui temos uma lista de todas as vacinas e seus correspondentes em Strings. E por causa dessa correspondência temos de criar mais algumas coisinhas nesse Enum. Primeiro um campo String nome, um construtor e dois métodos. O primeiro método serve para encontrarmos uma vacina justamente por uma String, que deve ser exatamente igual àquelas da lista. O segundo é um simples Getter do nome. A importância desses métodos vai ficar mais clara quando eu lhes mostrar a classes responsável para conversão de um Enum para um campo no banco de dados. Aqui está:

`/src/main/java/io/orangehealth/bvac/domain/VaccineConverter.java`

```
@Converter(autoApply = true)
public class VaccineConverter implements AttributeConverter<Vaccine, String> {

	@Override
	public String convertToDatabaseColumn(Vaccine vaccine) {
		return vaccine.getName();
	}

	@Override
	public Vaccine convertToEntityAttribute(String dbData) {
		return Vaccine.findByVaccineName(dbData);
	}
}
```

Esta classe é responsável pela conversão, ela implementa a interface AttributeConverter, aqui você encontra um breve exemplo de como fazer isso: https://www.baeldung.com/jpa-persisting-enums-in-jpa. Além disso ela tem a anotação @Converter, com ela a JPA aplica automaticamente as conversões, uma vez feita as sobrecargas dos métodos convertToDatabaseColumn e convertToEntityAttribute. Agora o problema introduzido pelas enumerações está resolvido.

Voltando à entidade, em data de aplicação temos um esquema parecido com a data de nascimento em usuário.

Em usuário temos o outro lado da moeda daquele @OneToMany. Aqui quem aparece é o @ManyToOne, que indica uma relação bidirecional. Você encontra um material muito bom sobre essas duas anotações aqui nesse link: https://www.baeldung.com/hibernate-one-to-many. Aqui temos de sobrescrever o padrão dessa anotação, precisamos de um FetchType.EAGER uma vez que estamos carregando duas tabelas de uma vez só. Mais uma vez a @JsonIgnoreProperties é usada para a prevenção de erros no carregamento da entidade.

Agora sobre os construtores, podemos ver que existem três deles. O padrão protegido para que não sejam criados objetos vazios. Um segundo que não incluí um usuário. E finalmente um terceiro que inclui todos os campos.

Você deve ter reparado que deixei um “to do” alí no meio do código. Isso foi de propósito. Quando falamos de persistência de dados e ORM(Object-relational mapping) na mesma conversa deve se fazer menções a algumas problemáticas, e uma delas é sobre a igualdade dos objetos. Decidi não entrar muito nessa assunto pois é um oceano de informação. Basicamente cada objeto tem uma espécie de impressão digital única na JVM, além de uma outra impressão digital criada pelo framework ORM que você usa e essas duas não batem, causado problemas quando você carregada um objeto na memória, quando você atualiza ou faz operações em objetos que já foram salvos. Um ótimo artigo sobre isso, e que ainda propõe uma solução(!!!) para isso é este que deixei no código. Por favor dê uma olhada aqui: https://web.archive.org/web/20170524072455/http://www.onjava.com/pub/a/onjava/2006/09/13/dont-let-hibernate-steal-your-identity.html?. E justamente, umas das resoluções do autor passa por sobrescrever aqueles dois métodos.

Depois disso apenas Getter e Setters.

O próximo passo agora é a criação dos repositórios. Aqui o trabalho é bem simples, e para as nossas necessidades é suficiente que a classe “estenda” apenas de CrudRepository, uma vez que seus métodos todavia básicos são suficientes para nossa aplicação. Dê uma olhada:

`/src/main/java/io/orangehealth/bvac/repository/UserRepository.java`

```
/**
 * User Repository
 *
 * @author Rafael Rodrigues
 */

public interface UserRepository extends CrudRepository<User, Long>{
	Optional<User> findByEmail(String email);
	Optional<User> findByCpf(String cpf);
}
```

`/src/main/java/io/orangehealth/bvac/repository/VaccineInjectionRepository.java`

```
/**
 * Vaccine Injection Repository
 *
 * @author Rafael Rodrigues
 */

public interface VaccineInjectionRepository extends
	CrudRepository<VaccineInjection, Long> {
}
```

Sobre o repositório de aplicação de vacinas não há muito o que ser dito, ele é bem simples. Já no repositório de usuários temos dois query methodos, uma para encontrar um usuário pelo e-mail e outro para encontrá-lo pelo CPF, esses dois métodos serão utilizados pelos serviços de usuário. Vamos então dar uma olhada nos serviços de nossa aplicação:

`/src/main/java/io/orangehealth/bvac/service/UserService.java`

```
/**
 * User Service
 *
 * @author Rafael Rodrigues
 */

@Service
public class UserService {
	private UserRepository userRepository;

	@Autowired
	public UserService(UserRepository userRepository) {
		this.userRepository = userRepository;
	}

	/**
	 * Creates and saves a user
	 *
	 * @param User signupUser
	 * @return Optional of user, empty if already exists
	 */
	public Optional<User> signup(User signupUser) {
		Optional<User> user = Optional.empty();
		if (!userRepository.findByEmail(signupUser.getEmail()).isPresent()
		&& !userRepository.findByCpf(signupUser.getCpf()).isPresent()) {
			user = Optional.of(userRepository.save(new
```

```
                                                User(signupUser.getFirstName(),
                                                      signupUser.getLastName(),
                                                      signupUser.getEmail(),
                                                      signupUser.getCpf(),
                                                      signupUser.getBirthDate())));
            }
            return user;
      }
}
```

Neste serviço temos o método que cria um usuário no nosso banco de dados, é o método signup. O que ele faz é primeiro inicializar um objeto que aponta para um Optional vazio. Logo em seguida temos um a condição, leia-se: “Se não (notem a exclamação dentro dos parênteses do if) existe um usuário cadastrado com o e-mail passado nos parâmetros do método signup e se não existe um usuário cadastrado com o CPF passado como parâmetro, faça...”. O que vem dentro do if é justamente preencher esse Optional vazio com os dados passados do usuário que quer se cadastrar. Caso essa condição não seja satisfeita, o método retorna aquele mesmo Optional vazio de antes. É isso. Agora o outro serviço:

`/src/main/java/io/orangehealth/bvac/service/VaccineInjectionService.java`

```
/**
 * Vaccine Injection Service
 *
 * @author Rafael Rodrigues
 */

@Service
public class VaccineInjectionService {
      private UserRepository userRepository;
      private VaccineInjectionRepository vaccineInjectionRepository;

      @Autowired
      public VaccineInjectionService(UserRepository userRepository,
                  VaccineInjectionRepository vaccineInjectionRepository) {
            this.userRepository = userRepository;
            this.vaccineInjectionRepository = vaccineInjectionRepository;
      }

      /**
       * Creates and saves a vaccine injection
       *
       * @param vaccineName
       * @param injectionDate
       * @param cpf
       * @return Optional of vaccine injection, empty if user don't exist
       */
      public Optional<VaccineInjection> register(Vaccine vaccineName,
                                          LocalDate injectionDate, String cpf) {
            Optional<VaccineInjection> vaccineInjection = Optional.empty();
            if (userRepository.findByCpf(cpf).isPresent()) {
                  vaccineInjection = Optional.of(vaccineInjectionRepository
                              .save(new VaccineInjection(vaccineName,
                                          injectionDate,
                                          userRepository.findByCpf(cpf).get())));
            }
            return vaccineInjection;
      }
}
```

Agora neste serviço de cadastro de aplicação de vacinas nos temos o método register. O que ele faz, bem parecido com o signup é criar um registro de vacinação. Ele recebe três parâmetros: um nome de vacina, um data de aplicação e um CPF como String. Mais uma vez, cria um Optional de aplicação de vacina, para depois entrarmo em um if que se lê: “Se há (notem que aqui não há negação) um usuário como o CPF especificado nos parâmetros, então…”. Dentro do if, com a ajuda de duas dependências “autowired”, o repositório de usuários e de aplicação de vacinas, criamos um registo de vacinação e salvamos no nosso banco de dados.

Agora vamos à camada Web:

`/src/main/java/io/orangehealth/bvac/web/UserController.java`

```
/**
 * User Controller
 *
 * @author Rafael Rodrigues
 */

@RestController
@RequestMapping(path = "/api/users")
public class UserController {
	@Autowired
	private UserService userService;

	/**
	 * Signup Endpoint
	 *
	 * @param User object
	 * @return HTTP status 201 if success, HTTP 400 if fails
	 */
	@PostMapping(path = "/signup")
	@ResponseStatus(HttpStatus.CREATED)
	public User singup(@Valid @RequestBody User user) {
		return userService.signup(user)
				.orElseThrow(() -> new
					ResponseStatusException(HttpStatus.BAD_REQUEST,
					"CPF or e-mail already registered."));
	}
}
```

Esse é o Controller de usuários, bem simples com apenas um endpoint. Ele tem como com path padrão “/api/users”. Nosso método está anotado com o @PostMapping com um path adicional de “/signup” e com a @ResponseStatus que determina qual será o status HTTP enviado para o cliente no sucesso do método, aqui um 201, created. Nosso método signup que recebe um usuário como parâmetro também é bem simples: ele faz um chamado do método do serviço de usuário e retorna o objeto que foi salvo, caso o retorno seja um Optional vazio ele retorna um ResponseStatusException, como é possível definir um status HTTP para o cliente, aqui no caso um 400, bad request, e uma mensagem de erro. Lembrando, quando o método do serviço retorna um objeto vazio é porque o usuário já foi cadastrado, então nossa mensagem de erro pode se apoiar nisso.

`/src/main/java/io/orangehealth/bvac/web/VaccineInjectionController.java`

```
/**
 * Vaccine Injection Controller
 *
 * @author Rafael Rodrigues
 */
```

```
@RestController
@RequestMapping(path = "/api/vaccineInjections")
public class VaccineInjectionController {
	@Autowired
	private VaccineInjectionService vaccineInjectionService;

	/**
	 * Vaccine Injection Registration Endpoint
	 *
	 * @param RegisterVaccination object
	 * @return HTTP status 201 if success, HTTP 400 if fails
	 */
	@PostMapping(path = "/register")
	@ResponseStatus(HttpStatus.CREATED)
	public VaccineInjection register(
		@Valid @RequestBody RegisterVaccinationDto registerVaccinationDto) {
		return vaccineInjectionService
				.register(registerVaccinationDto.getVaccineName(),
					registerVaccinationDto.getInjectionDate(),
					registerVaccinationDto.getCpf())
				.orElseThrow(() -> new
					ResponseStatusException(HttpStatus.BAD_REQUEST,
					"No user bind to this CPF registered."));
	}
}
```

Você já deve ter notado que em java se você sabe fazer uma coisa, muito provável que você saiba fazer muitas outras da mesma forma, essa consistência é muito valiosa. Então mais uma vez, como no Controller de usuários temos aqui um Controller de aplicação de vacina, bem no mesmo caminho. Temos nossas anotações @RestController, @ResquestMapping definindo um path, aqui “/api/vaccineInjections”. Apenas um método anotado com @PostMapping definindoum path “/register” para o método e o método em si. Para facilitar a vida do usuário, ao invés de se passarem dois objetos inteiro como parâmetros, decidi criar um DTO(Data Transfer Object), vamos dar uma olhada nele agora:

```
/**
 * Register Vaccine DTO
 *
 * @author Rafael Rodrigues
 */
public class RegisterVaccinationDto {
	@NotNull
	private Vaccine vaccineName;
	@NotNull
	@DateTimeFormat(pattern = "yyyy-MM-dd")
	private LocalDate injectionDate;
	@NotNull
	@Pattern(regexp = "^\\d{3}\\.\\d{3}\\.\\d{3}\\-\\d{2}$")
	private String cpf;

	public RegisterVaccinationDto(Vaccine vaccineName,LocalDate injectionDate,
			String cpf) {
		this.vaccineName = vaccineName;
		this.injectionDate = injectionDate;
		this.cpf = cpf;
	}

	// Getters and Setters
}
```

Além de diminuir a banda larga usada nos envios e recebimentos via internet, o que pode ser pouco notado numa aplicação tão pequena mas que pode ser um pesadelo numa aplicação maior, outro benefício dos DTOs é encapsulamento mais rigoroso, abstraindo algumas informações dos usuários, deixando nossa API mais limpa e "userfriendly", o que justifica muito bem essa pequena repetição de código que ao meu ver não vai nada com o DRY. Nesse DTO temos apenas três campos, nome da vacina, data de aplicação e CPF. Como nas entidades, temos as mesmas condições e restrições para os campos, uso de Enum e regex.

Voltando ao Controller. Nosso método recebe o dito DTO, faz um chamado do método register do serviço de aplicação de vacina e retorna um objeto do tipo VaccineInjection, passando um status HTTP pro cliente de 201, created. Caso o método falhe, mais uma vez usamo o ResponseStatusException, retornando pro cliente um status 400, bad request, e uma mensagem. Lembrando que quando o método do serviço falha é porque a busca por CPF não encontrou usuário nenhum como este CPF, então nossa mensagem de erro pode se apoiar nisso.

Se você chegou até aqui, você já tem uma baita de uma aplicação funcionando. Mas como prometido ainda falta fazer a alteração do banco de dados para finalizarmos. Isso é bem simples de ser feito, uma boa referência fica aqui neste link: https://spring.io/guides/gs/accessing-data-mysql/. Aqui no nosso caso basta que removamos a nossa dependência H2 e adicionemos o driver do MySql no seu pom.xml. Ele deve ficar com uma cara parecida com essa:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.3</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
```

```
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Agora, algumas poucas configurações no seu arquivo application.properties:

`/src/main/resources/application.properties`

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=seuUsusario
spring.datasource.password=suaSenha
server.error.include-message=always
server.error.include-binding-errors=always
```

As primeiras quatro linhas são as configurações necessárias para tudo rodar bem, as duas últimas são necessárias para que as mensagens de erros das entidades apareçam no lado do cliente. E pronto. Simples assim!

E com isso terminamos. Mas não para por aqui, temos muitos outros próximos passos. Por exemplo resolver o problema de igualdade dos objetos já mencionado. Mas há uma outra coisa de muitíssimo importância: testes. Nunca se esqueça de testar sua aplicação. Vou lhes dar um gostinho de como isso poderia ser feito logo. Uma boa pedida é seguir as diretrizes do TDD(Test-Driven Development). Outra coisa seria inserir uma camada de segurança na aplicação. E a lista continua.

Sobre os testes, aqui vai um teste integrado do Controller de usuários:

`/src/test/java/io/orangehealth/bvac/web/UserControllerTest.java`

```
/**
 * User Controller tests
 *
 * @author Rafael Rodrigues
 */
@WebMvcTest(UserController.class)
class UserControllerTest {

      private Optional<User> testUser = Optional
                  .of(new User("Malcolm", "X", "malcolmx@email.com",
                              "000.000.000-00", LocalDate.of(1925, 05, 19)));
      private String jsonContent = "{\"firstName\": \"Malcolm\","
                                             + "\"lastName\": \"X\","
                                             + "\"email\": \"malcolmx@email.com\","
                                             + "\"cpf\": \"000.000.000-00\","
                                             + "\"birthDate\": \"1925-05-19\"}";

      @Autowired
```

```
	private MockMvc mockMvc;

	@MockBean
	private UserService userService;

	@Test
	void testSingupHappyPath() throws Exception {
		Mockito.when(userService.signup(ArgumentMatchers.any(User.class)))
			.thenReturn(testUser);
		this.mockMvc.perform(MockMvcRequestBuilders.post("/api/users/signup")
				.content(jsonContent)
				.contentType(MediaType.APPLICATION_JSON))
				.andDo(MockMvcResultHandlers.print())
				.andExpect(MockMvcResultMatchers.status().isCreated());

	}

	@Test
	void testSignupUnhappyPath() throws Exception {
		Mockito.when(userService.signup(testUser.get())).thenReturn(testUser);
		this.mockMvc.perform(MockMvcRequestBuilders.post("/api/users/signup")
				.content(jsonContent)
				.contentType(MediaType.APPLICATION_JSON))
				.andDo(MockMvcResultHandlers.print())
				.andExpect(MockMvcResultMatchers.status().isBadRequest());
	}
}
```

Sei que não muito provavelmente esse não é o melhor jeito de se testar um Controller, mas serve como exemplo. Ainda estou fazendo minhas pesquisas e aprendendo, espero que esse exemplo lhe sirva de inspiração para fazer o mesmo.

Muito obrigado se você chegou até aqui. Foi um desafio muito frutífero a construção dessa aplicação. Meu nome é Rafael Rodrigues e esse foi meu post de hoje. Até a próxima!