Objetivo

O objetivo desse texto é explicar como implementar uma API REST para controlar a vacinação da população brasileira.

A API a ser desenvolvida deve conter os seguintes endpoints capazes de realizar:

- O Cadastro de usuário com os campos nome, e-mail, cpf e data de nascimento, sendo todos obrigatórios. E os campos e-mail e cpf devem ser únicos na tabela que armazena os usuários;

- O Cadastro de aplicação de vacina com os campos data da aplicação, o nome da vacina aplicada e uma referência para o usuário que recebeu a vacina, nesse caso a tabela que armazena as aplicações de vacina irá conter uma chave estrangeira para a tabela que armazena os usuários;

Essa API será desenvolvida com tecnologias da plataforma Java, tais como Spring Boot, JPA e Hibernate.

Implementação

Model

Vamos começar a implementação pela camada model. Ela é responsável por representar as informações que serão persistidas no banco de dados, no nosso caso o cadastro de usuários e de aplicações de vacinas.

As classes dessa camada serão POJOs contendo anotações da JPA. Mas o que seria JPA? É a sigla para Java Persistence API, uma especificação do mundo Java que determina como deve funcionar a persistência de dados em uma aplicação.

Por ser apenas uma especificação, a JPA precisa de um provedor, um framework que a implemente e de fato faça valer o que ela determina. Para o desenvolvimento da API REST será utilizado o Hibernate, um framework que faz o mapeamento objeto relacional.

O Hibernate, sendo capaz de mapear classes da aplicação para tabelas do banco, tira do desenvolvedor(a) a responsabilidade de “desmontar” um objeto para ser salvo no banco ou “montar” um objeto a partir do retorno de uma consulta ao banco, permitindo assim que o código da aplicação se mantenha trabalhando a nível de objeto.

A imagem a seguir demonstra o uso de anotações da JPA no mapeamento de uma entidade:

```java
@Entity
public class Usuario {

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

    private String nome;

    @Column(unique = true)
    private String email;

    @Column(unique = true)
    private String cpf;

    private LocalDate dataNascimento;
}
```

O @Entity indica que essa classe representa uma tabela no banco de dados e como nenhum nome foi definido será criado no banco automaticamente uma tabela chamada usuario.

As anotações @Id e @GeneratedValue indicam que o atributo em questão representa a chave primária da tabela e define também a forma como serão gerados os valores dessa chave, respectivamente.

E a anotação @Column serve para configurar alguns parâmetros da coluna da tabela referenciada pelo o atributo da classe. Usando *unique* estamos dizendo que os valores de e-mail e cpf devem ser únicos.

Durante esse mapeamento é importante dar preferência às anotações da JPA para que em caso de mudança de provedor, essa mudança seja mais tranquila.

Repository

O próximo passo é implementar a camada repository. Ela é a responsável por fazer o acesso aos dados, seja lendo ou escrevendo no banco. Para implementá-la usaremos uma módulo do Spring Boot chamado Spring Data JPA.

O Spring Data JPA torna muito simples o acesso ao banco de dados por meio de consultas que são de praxe quando estamos manipulando registros. Outro recurso interessante é a criação de consultas baseadas nos nomes dos métodos. Para ficar mais claro vamos ver a imagem a seguir:

```java
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Usuario findByCpfOrEmail(String cpf, String email);

    Usuario findByCpf(String cpf);
}
```

Na imagem a gente tem a interface UsuarioRepository herdando a interface JpaRepository. É importante perceber que esta última usa generics e que precisamos indicar a entidade que será manipulada bem como o tipo da chave primária.

Feito isso o Spring irá conseguir gerar, em tempo de execução, a implementação dos métodos dessa interface. E graças ao Spring Data JPA, o Spring também irá conseguir gerar as consultas referentes ao métodos *findByCpfOrEmail* e *findByCpf* já que estes seguem a convenção de nomes do Spring Data JPA.

Por exemplo para o método *findByCpf* será gerada uma consulta semelhante a:

*SELECT * FROM USUARIO WHERE CPF = ?*

OBS.: Um detalhe importante e que eu não havia mencionado ainda diz respeito a configuração de acesso ao banco de dados e definição do hibernate como provedor JPA. Sem isso as camadas model e repository não funcionarão.

Antes do Spring Boot essa configuração era feita por meio de arquivos XML ou de maneira programática usando classes Java. Agora essas configurações podem ser feitas em arquivos .properties, o que facilita até o uso de profiles, ou seja, podemos ter um arquivo para cada ambiente em que a nossa aplicação irá rodar, como por exemplo desenvolvimento, homologação e produção.

Toda aplicação Spring Boot já nasce com um arquivo de configuração que é o application.properties, a seguir podemos ver como ficou o da API que estamos desenvolvendo:

```
# data source
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:vacinacao
spring.datasource.username=sa
spring.datasource.password=

# jpa
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update

# h2
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Nas configurações referentes a datasource configuramos as credenciais de acesso ao banco. Já nas configurações referentes a JPA estamos definindo o Hibernate como provedor e que sempre que uma entidade mudar ele deverá fazer refletir no banco essa mudança(Isso não é uma boa prática, feita aqui apenas para teste).

Algo interessante que o Spring Boot faz para a gente e que demonstra a preocupação da ferramenta em agilizar o desenvolvimento, configurando automaticamente algumas coisas, é que ao habilitarmos o Spring Data JPA já será criada uma implementação da técnica Open Session in View.

Em aplicações que usam JPA/Hibernate nós precisamos nos preocupar em obter um EntityManager para a partir daí conseguir manipular as entidades e sempre lembrando de fechá-lo após as operações, o que é algo um pouco trabalhoso para quem está desenvolvendo.

Então por meio de um filter o Spring Boot implementa a técnica Open Session in View preparando um EntityManager quando uma requisição chegar na API REST e fechando-o na resposta da requisição.

Service