Passo a passo da implementação do desafio

Para construir API do desafio utilizei as seguintes tecnologias do ecossistema Spring :

- Spring Data JPA
- Spring MVC
- Spring Boot

Sabendo das tecnologias referente ao ecossistema do Spring que estão envolvidas nesse projeto , estarei apresentando melhor onde utilizei cada uma para resolução do desafio.

CRIAÇÃO DO PROJETO

Para criar projeto utilizamos Spring Boot , é um módulo que compoẽm o ecossistema do Spring cuja finalidade facilitar a criação de projetos com base na sua visão opinativa e com isso já realizar autoconfiguração.

- Benefícios :
    - Autoconfiguração
    - Redução de código de configuração
    - Padronização na criação dos projetos
    - Possibilidade Modificar alguma configuração pré-estabelecida

Primeiramente antes de falar das classes estarei mostrando algumas configurações iniciais.Sabemos que nossa API irá se conectar em algum banco dados para persistir os dados , então precisamos configurar algumas variáveis para que nossa API consiga interagir com banco de dados.

Como estamos utilizando Spring Boot ele facilita bastante nossa vida, consequentemente fazer nossa API conectar ao banco de dados será bem simples basta ir até arquivo chamado application.properties um arquivo criado pelo Spring Boot onde podemos especificar configurações necessárias.

```
spring.jpa.database=MYSQL
spring.datasource.url=jdbc:mysql://localhost/controle_vacinas?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username =root
spring.datasource.password = root
```

Como podemos observar acima, agora nossa API está apta a conectar com banco de dados que nosso caso é MySQL.

CAMADA DE MODELO E REPOSITORY

- Spring Data JPA
  - Basicamente utilizei Spring Data JPA para trabalhar com camada persistência e acesso a dados na qual o Spring Data JPA facilita a criação de repositórios JPA e quando digo que facilita , facilita mesmo ,com poucas linhas de códigos conseguimos interagir com banco de dados de forma simples.
  - Nessa API teremos as seguintes classes modelo :
    - Usuário
    - Vacina
    - Visto sobre classes necessárias , estarei apresentando os atributos de ambas classes que foram solicitado no desafio
      - Usuário(código , nome , cpf , email , data nascimento)
      - Vacina(código , nome , usuário , data aplicação)

```
@Entity
@Table(name = "tb_usuario")
public class Usuario {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;
@NotBlank
private String nome;

@NotBlank
@CPF
private  String cpf;

@NotBlank
@Email
private String email;

@Column(name = "data_nascimento")
private Date dataNascimento;
```

No código acima referente às entidades Usuário , Vacina e seus respectivos atributos usei algumas anotações da JPA para fazermos mapeamento objeto relacional e outras anotações da Bean Validation para validar a entrada da nossa API como por exemplo não aceitar valor nulos , vazios , validação do cpf e do e-mail

```
  @ManyToOne
    private Usuario usuario;
```

A anotação referente a JPA @ManyToOne presente na entidade Vacina ,basicamente existe um relacionamento entre Usuário e Vacina ,nesse caso podemos entender que um determinado usuário pode receber várias vacinas ou seja um único usuário pode está ligado a várias vacinas.

Agora sabendo quais entidades precisamos para conseguir persistir , recuperar , deletar e atualizar essas entidades criei uma interface ou repositório na qual estende outra interface chamada JpaRepository que contém uma série de métodos para trabalharmos com operações CRUD em cima dessas entidades que mencionei.

```
public interface VacinaRepository  extends JpaRepository<Vacina , Long> {
}
```

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

    @Query("SELECT u FROM Usuario u WHERE u.email = ?1 ")
    public Usuario findByEmail(String email);

    @Query("SELECT u from Usuario u WHERE u.cpf = ?1")
    public Usuario findByCpf(String cpf);

}
```

Podemos observar acima temos dois repositórios, um para entidade vacina e outro para usuário.Precisamos estender outra interface no JpaRepository que utiliza generics onde passamos qual entidades refere-se aquele repository e também qual tipo de dados que configuramos lá na nossa entidade referente código neste caso foi um Long , agora essa interface JpaRepository sabe o que vamos salvar , por exemplo o método buscar pelo ID ou código sabe que será um Long.

Visto sobre nosso modelo e os repositórios, estarei apresentando agora as camadas service e controller .

CAMADA CONTROLLER

- Como foi requisitado no desafio é preciso ter uma Api em forma webservice , então teremos que receber requisições HTTP.
- Então utilizei o Spring MVC para podermos receber essas requisições e também poder usufruir uma série de funcionalidades que Spring MVC fornece.

- Criamos duas classes controladoras nesse caso UsuarioController ,e VacinaController.

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

   @Autowired
   private UsuarioService usuarioService;

   @PostMapping
   public ResponseEntity<Usuario> criar(@Valid @RequestBody  Usuario usuario) {
        Usuario usuarioSalvo = usuarioService.salvar(usuario);

    return ResponseEntity.status(HttpStatus.CREATED).body(usuarioSalvo);

}
```

@RestController essa anotação facilita o retorno , ou seja retorna o objeto e os dados do objeto na qual são gravados diretamente na resposta HTTP como JSON ou XML.

@Request mapping é uma anotação que usei para definir uma rota na nossa API , ou seja quando for feita uma requisição para /usuarios ou para /vacinacao a classe referente ao controller que irá atender as requisições.

Usei o recurso que chamamos de Injeção de dependência do Spring para que ele possa injetar os nossos repositórios e camada serviço nesse caso optei por injetar repositório camada de serviço pois temos que aplicar regras de negócios que foram requeridas no desafio antes de ser salva no banco de dados.
Ao invés de deixar tudo dentro do controller basicamente fiz separação de responsabilidade o controller não precisa conhecer as de negócios ele simplesmente recebe os dados que foram passado na requisição e encaminha para service que é camada que conhece as regras de negócios dependendo se atender a

regra de negócio o service se comunica com controller para fazer o retorno de uma devida resposta de sucesso ou de falha etc.

Antes de apresentar a camada service de fato vamos ver mais algumas anotações importantes que usei :

@PostMapping bom como estamos utilizando o modelo arquitetural REST como sabemos possui uma série de regras definidas que devemos seguir para dizer de fato que nossa aplicação segue o modelo arquitetural REST. Essa anotação é referente ao protocolo HTTP cuja finalidade é especificar que estamos criando ou salvando um determinado recurso no caso usuário ou vacina.

Então como estamos seguindo REST é preciso utilizar verbo HTTP correto para alguma ação que iremos realizar.

Outra anotação importante é @RequestBody que vem no parâmetro do método em si , essa anotação basicamente fala pro Spring olha eu quero que você pega o que está vindo no corpo de requisição e associa esse Objeto Java,o que chamamos desserialização.

Importante explicar como Spring sabe hora de executar o método mapeado com PostMapping é importante ressaltar que poderíamos ter n métodos mapeados com outras anotações referente aos verbos HTTP.Mais de forma simples funciona da seguinte forma :

- Quando fazemos uma requisição por exemplo para /usuarios na qual foi especificado que seria verbo Post o Spring entende que quando chegar uma requisição para /usuarios e for do tipo Post o método que irá atender é o criar no nosso caso.

```
@RestController
@RequestMapping("/vacinacao")
public class VacinaController {

   @Autowired
   private VacinaService vacinaService;

   @Autowired
   private ApplicationEventPublisher publisher;

   @PostMapping
   public ResponseEntity<Vacina> criar(@Valid  @RequestBody  Vacina vacina){
     Vacina vacinaSalva = vacinaService.salvar(vacina);

        return ResponseEntity.status(HttpStatus.CREATED).body(vacinaSalva);
   }
```

No desafio foi pedido que se as propriedades dos objetos usuario e vacina , não poderia ser null ou vazio , para validar a entrada da nossa API utilizamos Bean Validation na qual já até mostrei.Mais só anotar lá na classe de modelo não irá para

funcionar precisamos utilizar @Valid para dizer olha antes de passar esse objeto para próxima camada verifique se os campos estão válidos se caso estiver ele segue o fluxo senão ele lança uma exceção na qual já vou falar como eu fiz para tratar as exceções.

Antes de mostrar a camada service preciso falar sobre status code HTTP ou seja são código de resposta que servem para dizer se ocorreu com sucesso , falhou , não encontrou um determinado recurso , se foi criado entre outras coisas.No nosso caso foi pedido se caso o recurso passar nas validações e atender as regras de negócios retorna o código 201(CREATED - o recurso foi criado com sucesso)caso contrário retorna 400(BAD REQUEST- uma requisição ruim) , por isso utilizei ResponseEntity para conseguirmos manipular resposta da requisição.E por fim a gente pega objeto que acabou de ser salvo no banco de manda na resposta também.

Quando estamos seguindo modelo arquitetural REST , é preciso utilizar os status code HTTP da maneira correta também.

CAMADA SERVICE

Estarei apresentando agora código referente a camada de serviço , só que antes preciso falar sobre quais são as regras que precisamos obedecer antes de cadastrar um usuário ou uma vacina no banco de dados :

- Não pode existir usuários com mesmo e-mail e cpf ou seja deverá ser único
- E para efetivamente cadastrar uma vacina precisamos verificar se o usuário existe de fato .Como mencionei lá no começo do artigo vacina tem vínculo com usuário ou seja para uma vacina ser cadastrada ela precisa de um usuário só que não faz sentido cadastrar uma vacina sem usuário então vamos validar antes de criar uma vacina se usuário existe de fato ou não

Dito sobre as regras de negócio vamos ver os códigos como ficaram

```
@Service
public class UsuarioService {

   @Autowired
   private UsuarioRepository usuarioRepository;

   public Usuario salvar(Usuario usuario){

       verificarExistencia(usuario.getEmail() , usuario.getCpf());
       usuario.setDataNascimento(new Date());
       return usuarioRepository.save(usuario);
   }

   private void verificarExistencia(String email , String cpf){
```

```
        Usuario usuarioEncontradoPeloEmail = usuarioRepository.findByEmail(email);
        Usuario usuarioEncontradoPeloCpf = usuarioRepository.findByCpf(cpf);

        if(usuarioEncontradoPeloEmail != null || usuarioEncontradoPeloCpf !=null){
            throw new UsuarioComEmailOuCpfJaExistenteException("O e-mail ou cpf que
foi informado já existe nossa base dados");
        }
    }

}
```

Essa é camada de serviço referente ao Usuário, então antes de salvar chamamos o método verificarExistenciaEmailCpf ,na qual passamos e-mail e o cpf do usuário que de fato estamos querendo salvar , aí método irá realizar duas consultas uma para email e outra para cpf . Aí depois aplicamos uma condicional para validar se realmente já existe algum usuário cadastrado que tenha mesmo email ou cpf e até mesmo os dois em comum.Se caso for verdadeiro irá lançar uma exceção que criamos chamada `UsuarioComEmailOuCpfJaExistenteException` .Caso for igual null significa que não existe aí método salvar contínua fluxo salva de fato o usuário.

```
@Service
public class VacinaService {

    @Autowired
    private VacinaRepository vacinaRepository;

    @Autowired
    private UsuarioRepository usuarioRepository;

    public Vacina salvar(Vacina vacina){
        Optional<Usuario> usuario =
usuarioRepository.findById(vacina.getUsuario().getCodigo());

        if(!usuario.isPresent()){
            throw  new UsuarioInexistenteExeption("Não podemos vincular esse usuário
a essa vacina pois o usuário não existe");
        }

        vacina.setUsuario(usuario.get());
        vacina.setDataAplicacao(new Date());
        return  vacinaRepository.save(vacina);
    }

}
```

No caso de Vacina a gente precisa injetar os dois repositórios para antes de salvar uma vacina e vincular um usuário teremos que realizar uma consulta simples na tabela referente a usuários para saber se existe ou não.

Então utilizamos o método findById em cima de usuário passando o código do usuário que recebemos para checar existência . Caso o usuário não esteja presente

retorna uma exceção que criamos chamada `UsuarioInexistenteExeption` caso o usuário realmente exista ele atribui usuário e salvar a vacina.

Alguns detalhes que não mencionei são referentes à anotação @Service que basicamente designa que classes serão um bean gerenciado pelo Spring o que nos possibilitar aplicar a injeção de dependência .Mas possui significado que é simplesmente definir que essas classes são referente a camada de serviço.

EXCEÇÕES

Para tratarmos os possíveis erros que teremos em relação às regras de negócios e também associado às anotações de validação, criei as seguintes classes:

- ApiException
- ApiExceptionHandler

E associado a camada service mais especificamente as regras de negócios foram necessários as seguintes exception :

- UsuarioComEmailOuCpfJaExistenteException
- UsuarioInexistenteExeption

A classe ApiException é uma classe que criei para representar um erro ou seja o que é relevante na hora mostra erro para cliente ou desenvolvedor saber sobre erro vejamos estrutura dessa classe :

```
public class ApiException {

   private String mensagemUsuario;
   private String mensagemDesenvolvedor;
   private ZonedDateTime timesStamp;
}
```

Temos um atributo específico para personalizar uma mensagem para usuário , desenvolvedor e mostrar data e hora na qual devido erro aconteceu.Essa classe usaremos lá no nosso ApiExceptionHandler.

Criei ApiExceptionHandler para não precisar tratar exceções explicitamente nos métodos do controlador pois isso é uma boa prática de desenvolvimento.

O tipo de tratamento que optei em usar é tratamento de exceções globais ou seja a ApiExceptionHandler enxergar toda nossa aplicação por isso anotamos com @ControllerAdvice com isso criamos manipulador padrão para *qualquer* exceção.

```
@ControllerAdvice
public class ApiExceptionHandler extends
ResponseEntityExceptionHandler {
```

}

Nós estendemos a `ResponseEntityExceptionHandler` porque possui uma série de métodos já prontos , nesse caso são métodos padrões.

Agora que falei sobre o motivo de se criar ApiExceptionHandler estarei mostrando o código como ficou dentro dessa classe que estamos manipulando três exceções :

```
1- handleMethodArgumentNotValid
2 - UsuarioComEmailOuCpfJaExistenteException
3 - UsuarioInexistenteExeption
```

A handleMethodArgumentNotValid é uma exceção relacionada ao bean validation sobre aquelas anotações @NotBlanck , @Cpf e @email.Ou seja @Valid lá no controller faz devidas validações caso falhe essa exceção que é lançada ou seja traduzindo seria “O argumento passado para método não é válido" , com isso ela executa a seguinte código caso venha falha :

```
@Override
protected ResponseEntity<Object>
handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders
headers,
                                                                HttpStatus status,
WebRequest request) {

   List<ApiException> erros = criarListaErros(ex.getBindingResult());

   return handleExceptionInternal(ex , erros , headers , HttpStatus.BAD_REQUEST ,
request);
}
```

É importante ressaltar que esse método foi sobrescrito então por padrão ele já tem configurado mensagem que irá retornar entre outras coisas. Só que mensagem muito verbosa então para simplificar mensagem vamos personalizar esse método na qual podemos mostrar mensagem da forma que queremos de fato . Então no retorno configuramos o corpo que de fato é a exceção que foi lançado , mensagem ou seja corpo , cabeçalho , status code e um web request.Lembrando que no

desafio foi pedido caso algum problema relacionado a entrada nos campos da API retorna status code 400(significa uma requisição ruim feita pelo cliente).

Um detalhe importante foi no body passar uma lista referente ao código abaixo.

```
private List<ApiException> criarListaErros(BindingResult bindingResult) {
   List<ApiException> erros = new ArrayList<>();

   for(FieldError fieldError : bindingResult.getFieldErrors()){
       String mensagemUsuario = messageSource.getMessage(fieldError ,
LocaleContextHolder.getLocale());
       String mensagemDesenvolvedor = fieldError.toString();
       erros.add(new ApiException(mensagemUsuario , mensagemDesenvolvedor,
HttpStatus.BAD_REQUEST , ZonedDateTime.now()));
   }

   return erros;
}
```

Optei por criar uma lista porque quando ocorreu um erro relacionado a um dos campos da API conseguimos mostrar qual campo que aconteceu o erro e uma mensagem explicando de fato o que acarretou no erro . Então o código acima faz basicamente isso, ele varre o objeto bindingResult verificar quais campos ocorreu erros joga na lista para que conseguimos mostrar no corpo da requisição os campos.

A gente vai setando os atributos que foram especificados lá no ApiException para formar a mensagem e adicionando na lista .

Importante ressaltar como configuramos a mensagem de fato:

```
@Autowired
private MessageSource messageSource;
```

Usamos essa interface do Spring na qual conseguimos nos comunicar com arquivos .properties e sobrescrever as respectivas mensagens que já é configurada por padrão pelo Bean Validation.

Nesse caso criei os seguintes arquivos :

- messages.properties

```
usuario.inexistente =  Usuário Inexistente para salvar vacina
usuario.duplicado-cpf-ou-email-ja-existente = CPF ou E-mail j\u00E1
existente



usuario.nome = Nome
usuario.email = E-mail
```

```
usuario.cpf = CPF
```

Basicamente segue estrutura de chave e valor , configuramos algumas coisas que queremos mostrar referente as exceções que criei entre outras e também personalizamos como queremos que os nomes das propriedades aparecem na mensagem.

- ValidationMessages.properties

```
javax.validation.constraints.NotBlank.message = {0} \u00E9 obrigat\u00f3rio(a)
javax.validation.constraints.Email.message = {0} inv\u00E1lido
org.hibernate.validator.constraints.br.CPF.message = O {0} informado est\u00E1
inv\u00E1lido
```

Esse arquivo também tem uma chave que possui um determinado valor, no caso a mensagem que será exibida.

Um detalhe: usamos o arquivo para sobrescrever as mensagens que são definidas por padrão pela Bean Validation.

Com isso, restou somente mostrar métodos referente as exceções que eu criei.

- UsuarioComEmailOuCpfJaExistenteException
- UsuarioInexistenteExeption

```
@ExceptionHandler({UsuarioComEmailOuCpfJaExistenteException.class})
protected ResponseEntity<Object> handleUsuarioComEmailJaExistenteException
(UsuarioComEmailOuCpfJaExistenteException ex,WebRequest request) {

   String mensagemUsuario = messageSource.getMessage("usuario.inexistente" , null ,
LocaleContextHolder.getLocale());
   String mensagemDesenvolvedor = ex.getMessage();
   List<ApiException> erros = Arrays.asList(new ApiException(mensagemUsuario ,
mensagemDesenvolvedor , HttpStatus.BAD_REQUEST , ZonedDateTime.now()));


   return handleExceptionInternal(ex , erros , new HttpHeaders(), HttpStatus.BAD_REQUEST ,
request);
}
@ExceptionHandler({UsuarioInexistenteExeption.class})
protected ResponseEntity<Object> handleUsuarioInexistenteExeption
(UsuarioInexistenteExeption ex,WebRequest request) {

   String mensagemUsuario = messageSource.getMessage("usuario.inexistente" , null ,
LocaleContextHolder.getLocale());
   String mensagemDesenvolvedor = ex.getMessage();
   List<ApiException> erros = Arrays.asList(new ApiException(mensagemUsuario ,
mensagemDesenvolvedor , HttpStatus.BAD_REQUEST , ZonedDateTime.now()));


   return handleExceptionInternal(ex , erros , new HttpHeaders(),
HttpStatus.BAD_REQUEST , request);
}
```

Estarei mostrando a classe:

```
public class UsuarioComEmailOuCpfJaExistenteException extends
RuntimeException {

   public UsuarioComEmailOuCpfJaExistenteException(String
mensagem){
        super(mensagem);
   }

   public UsuarioComEmailOuCpfJaExistenteException(String
mensagem , Throwable cause){
        super(mensagem , cause);
   }

}
```

```
public class UsuarioInexistenteExeption extends
RuntimeException {

   public UsuarioInexistenteExeption(String mensagem){
        super(mensagem);
   }

   public UsuarioInexistenteExeption(String mensagem ,
Throwable cause){
        super(mensagem , cause);
   }

}
```

Onde lá na camada service passamos a mensagem ou também passar a causa da exceção ,com isso finalizo a parte de exceções.

EXTRA

Gostaria antes finalizar o artigo fazer um adendo lá na camada controller o desafio pedia para que fosse implementado somente ação de criar ou salvar recurso.Só que estamos falando de REST é uma boa prática ao salvar recurso podermos mostrar localização ou seja endereço onde podemos acessar aquele recurso no nosso caso seria uma busca pelo código.Só que não irei mostrar essa implementação dessa ação de buscar um determinado recurso pelo código.Mais estarei mostrando um recurso do Spring chamado eventos.

```
@PostMapping
public ResponseEntity<Vacina> criar(@Valid  @RequestBody  Vacina vacina){
   Vacina vacinaSalva = vacinaService.salvar(vacina);

   URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
           .path("/{codigo}").buildAndExpand(vacinaSalva.getCodigo()).toUri();


   return ResponseEntity.created(uri).body(vacinaSalva);
}
```

O código acima além de salvar uma vacina ele configura o header location nesse caso imagina que estamos criando uma vacina cujo código gerado foi 1 , simplesmente o que irá acontecer é que no cabeçalho da requisição na área location irá aparecer o seguinte :

- https://localhost:8080/vacinacao/1

Basicamente criamos uma URL que nos permite chegar até o recurso de código 1 esse é o resultado do código.
Só que precisamos repetir isso para todos os nossos endpoint no caso para usuário também

```
@PostMapping
public ResponseEntity<Usuario> criar(@Valid @RequestBody  Usuario usuario) {
   Usuario usuarioSalvo = usuarioService.salvar(usuario);

   URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()
           .path("/{codigo}").buildAndExpand(usuarioSalvo.getCodigo()).toUri();


   returnResponseEntity.created(uri).body(usuarioSalvo);
}
```

Ou seja imagina que tenhamos 10 endpoint teríamos que fazer a mesma coisa , com isso o nosso código não está legal há muita duplicação.

Para resolver esse problema usaremos o recurso de eventos do Spring nesse caso criaremos uma classe chamada RecursoCriadoEvent que será estrutura do nosso evento ou seja o que precisamos para configurar header location:

- `HttpServletResponse`
- `Long codigo;`

```
public class RecursoCriadoEvent extends ApplicationEvent {

   private Long codigo;

   private HttpServletResponse httpServletResponse;

   public RecursoCriadoEvent(Object source , Long codigo , HttpServletResponse
response) {
```

```
        super(source);
        this.codigo = codigo;
        this.httpServletResponse = response;

    }
```

Essa será a estrutura do evento, precisamos estender de `ApplicationEvent` para dizer ao Spring que essa classe será representação de um evento que iremos disparar.

Agora precisamos criar Listener o cara que irá escutar esse evento de fato no caso pegar código e realizar configuração do header location pois até então não fizemos nada que configure isso.Estarei mostrando o listener desse evento :

```
@Component
public class RecursoCriadoListener implements
ApplicationListener<RecursoCriadoEvent> {

    @Override
    public void onApplicationEvent(RecursoCriadoEvent recursoCriadoEvent) {
            Long codigo = recursoCriadoEvent.getCodigo();
            HttpServletResponse httpServletResponse =
recursoCriadoEvent.getHttpServletResponse();

        adicionandoHeaderLocation(codigo, httpServletResponse);

    }

    private void adicionandoHeaderLocation(Long codigo, HttpServletResponse
httpServletResponse) {
        URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri()

.path("/{codigo}").buildAndExpand(codigo).toUri();

        httpServletResponse.setHeader("Location" , uri.toASCIIString());
    }
}
```

Anotamos essa classe como @Component para que seja gerenciada pelo Spring quando for lançado o evento e com isso o Spring saiba encontrar quem será responsável por atender ao evento.

Precisamos implementar a interface `ApplicationListener<RecursoCriadoEvent>` passando no generics como parâmetro o Evento , que essa classe irá atender e também precisamos sobrescrever, o método `onApplicationEvent` e especificar o código que queremos executar quando evento for lançado.Agora vamos lançar o evento.

```
@RestController
@RequestMapping("/vacinacao")
public class VacinaController {
```

```
    @Autowired
    private VacinaRepository vacinaRepository;

    @Autowired
    private VacinaService vacinaService;

    @Autowired
    private ApplicationEventPublisher publisher;

    @GetMapping
    public List<Vacina> listarTodasVacinas(){
        return  vacinaRepository.findAll();
    }
    @PostMapping
    public ResponseEntity<Vacina> criar(@Valid  @RequestBody  Vacina vacina
,                                       HttpServletResponse
httpServletResponse){
        Vacina vacinaSalva = vacinaService.salvar(vacina);

        publisher.publishEvent(new RecursoCriadoEvent(this ,
vacinaSalva.getCodigo() , httpServletResponse));
        return ResponseEntity.status(HttpStatus.CREATED).body(vacinaSalva);
    }

}
```

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {

 @Autowired
  private UsuarioService usuarioService;

  @Autowired
  private ApplicationEventPublisher publisher;

  @PostMapping
  public ResponseEntity<Usuario> criar(@Valid @RequestBody  Usuario usuario ,
                          HttpServletResponse httpServletResponse) {
    Usuario usuarioSalvo = usuarioService.salvar(usuario);

    publisher.publishEvent(new RecursoCriadoEvent(this , usuarioSalvo.getCodigo() ,
httpServletResponse));

    return ResponseEntity.status(HttpStatus.CREATED).body(usuarioSalvo);
  }

}
```

Usei a interface `ApplicationEventPublisher` na qual permite-nos lançar o evento , dentro do método salvar , utilizei método publishEvent passando o evento RecursoCriado , depois configurando o código o this referente a quem lança o evento no caso a classe UsuarioController e response para podermos adicionar header location.Com isso nosso código ficou simples e enxuto.

FINAL

Antes de finalizar o artigo gostaria deixar o link do repositório lá do GitHub referente ao código fonte do projeto do desafio para poderem dar uma olhada melhor no código.

[GITHUB] - https://github.com/RafaelAmaralPaula/controle-vacinas-api
[E-MAIL] - rafaelpaulajr@gmail.com

Rafael Amaral De Paula
Um abraço !!