Opa, pessoal tudo bom?

Neste blog vou ensinar para vocês o que aprendi ate o momento sobre o método API REST em JAVA, método que aprendi nestes últimos dez dias para responder um desafio da Zup Orange Talents.

Antes de começarmos quero que saibam que sou aberto para opiniões, correções e afins. Não sou nenhum expertise na área mas vou deixar abaixo tudo que consegui aprender e desenvolvi nestes últimos dias.

Sem mais delongas, vamos lá!

Vou deixar aqui abaixo o desafio para voces entenderem o nosso objetivo e depois ate onde chegamos!

Desafio :

CONTEXTO:
Você está fazendo uma API REST que precisará **controlar veículos de usuários**.

O **primeiro passo** deve ser a construção de um cadastro de usuários, sendo obrigatórios: nome, e-mail, CPF e data de nascimento, sendo que e-mail e CPF devem ser únicos.

O **segundo passo** é criar um cadastro de veículos, sendo obrigatórios: Marca, Modelo do Veículo e Ano. E o serviço deve consumir a API da FIPE (https://deividfortuna.github.io/fipe/) para obter os dados do valor do veículo baseado nas informações inseridas.

O **terceiro passo** é criar um endpoint que retornará um usuário com a lista de todos seus veículos cadastrados.

Você deve construir 3 endpoints neste sistema, o cadastro do usuário, o cadastro de veículo e a listagem dos veículos para um usuário específico.

No endpoint que listará seus veículos, devemos considerar algumas configurações a serem exibidas para o usuário final. Vamos criar dois novos atributos no objeto do carro, sendo eles:

1.) Dia do rodízio deste carro, baseado no último número do ano do veículo, considerando as condicionais:
Final 0-1: segunda-feira
Final 2-3: terça-feira
Final 4-5: quarta-feira
Final 6-7: quinta-feira
Final 8-9: sexta-feira

2.) Também devemos criar um atributo de rodízio ativo, que compara a data atual do sistema com as condicionais anteriores e, quando for o dia ativo do rodizio, retorna *true*; caso contrario, *false*.

Exemplo A: hoje é segunda-feira, o carro é da marca Fiat, modelo Uno do ano de 2001, ou seja, seu rodízio será às segundas-feiras e o atributo de rodízio ativo será *TRUE*.
Exemplo B: hoje é quinta-feira, o carro é da marca Hyundai, modelo HB20 do ano de 2021, ou seja, seu rodizio será às segundas-feiras e o atributo de rodízio ativo será *FALSE*.

- Caso os cadastros estejam corretos, é necessário voltar o Status 201. Caso hajam erros de preenchimento de dados, o Status deve ser 400.

- Caso a busca esteja correta, é necessário voltar o status 200. Caso haja erro na busca, retornar o status adequado e uma mensagem de erro amigável.

B.) SEU DESAFIO:

Dado que você fosse implementar esse sistema utilizando Java como linguagem e Spring + Hibernate como stacks de tecnologia fundamentais da aplicação: escreva um post de blog explicando de maneira detalhada tudo que você faria para implementar esse código (pense como se estivesse contando para alguém que não manja de programação) 

Bora começar !

Certo, agora que já sabem o desafio vamos começar a "codar" este carinha !

Utilizaremos a linguagem Java para o desenvolvimento desta API, caso não saibam API seria uma forma de conseguirmos manipular aplicações já existentes de maneira mais rápida e ágil sem ser necessário coarmos absurdos e conhecermos todo o código, em outras palavras seria um controle remoto universal.

Entendendo o meu raciocínio.

Primeiro vamos começar criando algumas classes, em outras palavras vamos criar nossos principais personagens neste programa. Dentre eles são:
Usuário -> Usuário que o desafio nos propôs com seus respectivos dados.
Carro - > Assim como o "Usuário", porem com seus respectivos dados.
Tela de Login -> Tela onde mostraria a entrada do sistema informando se o carro cadastrado pode ser utilizado no dia do rodizio.

Criando Cadastro de Usuário

Para criar o nosso usuário, vamos começar dando uma declaração de @Entity, declaração que significa para o banco de dados que estes valores a serem inseridos podem ser mapeados no banco de dados, em outras palavras seria uma placa indicando para o banco quando for querer encontrar o usuário onde ele esta.

Neste momento vamos criar nossa classe, neste caso Usuarioval, classe que será responsável por ter todos aqueles campos lá atrás que o desafio nos pediu : cpf,email,etc...
Antes de começarmos a declarar as variáveis, vamos lembrar que uma destas variáveis é muito importante ela será única no nosso banco de dados, algo como um CPF em nosso país. Esta variável será o id, responsável por ser única, sendo também nossa chave principal no banco. Se não entendeu, seria algo como através dela conseguirmos pegar tudo que quiséssemos assim como seu CPF detem os dados de seu nome, nasc e afins.
No método Spring podemos colocar outra anotação para variáveis como esta, portanto utilizaremos o @Id (bem sugestivo né?).
Agora que já temos ideia de algumas anotações do método spring vamos começar a declara nossas variáveis. Para adiantar o processo vou colocar para vocês aqui como ficou no meu programa:

```
@Entity /* entidade mapeada para tabela no banco de dados */
public class Usuarioval {
    @Id
    ///@GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(nullable = false,unique = true)
    private long cpf;
    @Column(nullable = false, length = 11)
    private String nome;
    @Column(nullable = false, length = 50,unique = true)
    private String email;
    @Column(nullable = false, length = 8)
    private Date nasc;
```

Acima vocês podem ver que para cada variável usamos algumas anotações de @Column, esta anotação serve para dizermos ao nosso banco de dados que será como a própria tradução insinua: "Coluna", ou seja no banco de dados terá uma coluna id,cpf,email,etc...

Obs: Nullable significa se pode ser vazio o campo, unique significa se seria único este campo.

Certo, terminamos de declarar nossa variáveis, mas agora temos que lembrar que declaramos elas mas não adicionamos nenhum método a elas, ou seja de que adianta declararmos elas se o sistema não saberá pega-las e usa-las? Vamos então realizar o famoso Getter and Setter, ou seja adicionarmos funções de Getter (pegar) e Setter (delegar).

Para poupar um certo tempo vou deixar novamente como ficou meu código.

```
public long getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public long getCpf() {
    return cpf;
}

public void setCpf(long cpf) {
    this.cpf = cpf;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public Date getNasc() {
    return nasc;
}

public void setNasc(Date nasc) {
    this.nasc = nasc;
}
```

Agora já temos nosso primeiro personagem deste programa FEITO !

Vamos agora para o nosso SEGUNDO PERSONAGEM, o CARRO !

Como você ja entendeu muitas coisas ali atras, vamos poupar alguns caracteres neste Blog e vou colar o código para voces do personagem CARRO, lembrando que as variaveis mudam ja que o carro não tem nome, cpf e afins. Se estiver com dúvida porque criei as variavéis lembre-se do que o desafio propos.

```
@Entity
public class Carrosinf {
```

```
        @Id
//        @GeneratedValue(strategy = GenerationType.IDENTITY)

        private Integer qtdecarro;
        @Column(nullable = false)
        private Integer marca;
        @Column(nullable = false)
        private Integer modelo;
        @Column(nullable = false, length = 11)
        private String versao;
        @Column(nullable = false, length = 50)
        private String valor;

    public Integer getQtdecarro() {
        return qtdecarro;
    }

    public void setQtdecarro(Integer qtdecarro) {
        this.qtdecarro = qtdecarro;
    }

    public Integer getMarca() {
        return marca;
    }

    public void setMarca(Integer marca) {
        this.marca = marca;
    }

    public Integer getModelo() {
        return modelo;
    }

    public void setModelo(Integer modelo) {
        this.modelo = modelo;
    }

    public String getVersao() {
        return versao;
    }

    public void setVersao(String versao) {
        this.versao = versao;
    }

    public String getValor() {
        return valor;
    }

    public void setValor(String valor) {
        this.valor = valor;
    }
}
```

Entenda que a variável qtdecarro teria a mesma funcao do nosso id do personagem USUARIO.

Criando nosso cenário, ambiente, Controller ou como quiser chamar.

Agora, ja temos 2 personagens criados, porém temos outras coisas para configurar. Nossos personagens não estão associados a nenhum ambiente, ou seja seria como jogar Romel e Julieta em uma sala vazia. Para não deixarmos nosssos personagens sem ambiente, vamos cria-los eles agora.
Normalmente os devs criam estas classes a seguir como XXXcontroler, entendo que este tal de controller seria em outras palavras o caminho, ambiente, o cenário de nossos personagens, ou seja, onde eles ficarão no nosso banco de dados.
Bora, lá.
Antes de começarmos a criar a classe, vamos usar a anotação `@RestController` para definirmos ela como nosso cenário. Uma outra anotação que iremos utilizar será o `@GetMapping`, esta anotação é responsável por colocarmos o caminho no nosso servidor / rede onde estará o cadastro de nosso usuário.

```
@RestController

 class                              Usuariostatus                                {



    @GetMapping(path                =                "/servidor/usuario/{id}")
    public ResponseEntity consultar (@PathVariable("id") Integer id) {
        return                                                              null

    }
}
```

Neste caso você pode perceber que que nosso Personagem que tiver o seu devido id, poderá ser encontrado no caminho `"/servidor/usuario/{id}" .`

Faremos isto também para o nosso personagem CARRO que ficara assim:

```
@RestController
    @RequestMapping("/servidor/carros")
public                  class                   Carroval                    {

    @GetMapping
    public              List<Carrosinf>             listar()                {
        return                                                          null;
    }
```

Você pode estar se questionando a anotação @RequestMapping, ela tem o mesmo principio da @GetMapping porem o @GetMapping seria uma maneira curta de utilizar o @RequestMapping.

Certo, neste momento já temos nossos 2 personagens prontos e nossos 2 cenários. Agora vamos configurar nosso acesso ao banco de dados, conhecido como application.properties.

Configurando nosso application.properties (acesso ao banco de dados).

Neste documento fica as configurações de acesso ao seu banco de dados, assim como podemos inserir algumas outras funções como comandos para que qualquer alteração feita no banco apareça no nosso console de desenvolvimento quando o ambiente estiver ativo. Abaixo fica a configuração que usamos:

```
spring.datasource.url = jdbc:mysql://localhost:3306/mysql?useTimezone=true&serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password = 1234
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Você pode ver que na primeira linha encontra-se os dados de conexão do banco e afins.
Na segunda linha fica a informação do usuário que estamos utilizando no banco e na terceira linha fica nossa senha.
Na terceira linha deixamos o jpa.hibernate.ddl-auto recebendo o tipo Update, ou seja em outras palavras o banco de dados estará atualizando seus metadados em tempo real, para que não haja aqueles problemas como tempo de conexão com o banco de dados expirado.
E por ultimo temos na quinta linha a função show-sql = true que estará responsável por qualquer alteração que seja feito em nosso banco de dados aparece em nosso terminal as mudanças realizadas.
Terminamos agora nesta configuração de acesso ao banco de dados, vamos agora começar a ultima parte a ponte que ligara o programa ao banco de dados, onde conseguiremos salvar os dados que queremos.

Criando nosso repositório, nossa ponte do arco-íris.

Vamos criar agora nosso repositório para que seja possível salvar os dados que queremos inserir no banco assim como para que os dados que criemos sejam visíveis para consulta.
Para agora, vamos criar não mais uma classe e sim uma interface.

Nossa interface será o responsável por fazer as consultas em nosso banco de dados. Vocês verão que usaremos a anotação @Repository, para declararmos ela como nosso repositório.
Bora la.

```
@Repository
public interface Carrorepositorio extends JpaRepository <Carrosinf,
Integer>                                                          {
}
    @Repository
     public   interface   Usuariorepositorio   extends   JpaRepository
<Usuarioval,                        Integer>                      {
}
```

Temos aqui nossos dois repositórios para nossos 2 personagens. Agora que já temos nosso repositório vamos ajeitar os nossos @RestController.

Ajeitando nosso @RestController

Neste momento vou tentar te explicar diferente, vou copiar o codigo e vamos entendendo ele juntos:

```
@RestController
    @RequestMapping("/servidor/carros")
public class Carroval {
    @Autowired
    //neste momento estamos atribuindo funcoees ao repositorio, se
deixarmos sem o autowired o programa retornaria null
    private Carrorepositorio carrorepositorio;

    @GetMapping
    public List<Carrosinf> listar() {
        return carrorepositorio.findAll();
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public  Carrosinf salvar(@RequestBody  Carrosinf carros){
//Request Body converte corpo da requisicao para objeto cliente no
caso.
        return carrorepositorio.save(carros);
    }
}
```

@Autowired -- Responsável por atribuir funções ao nosso repositório para que ele não retorne o valor Null.
@GetMapping-- Usamos a função List, função que Lista no caso o que pedirmos, mas vamos usar o .findAll() para que seja listado todos os carros que cadastramos.
@PostMappin -- Responsável por criar o método post em nosso banco, ou seja o método de inserção de dados nele.

@ResponseStatus-- Responsável por criarmos aquela resposta / padrão de protocolo HTTP que o desafio queria, neste caso o 201 CREATED.
@RequestBody-- Responsável por inserir no nosso método de inserção Post os dados no corpo do nosso personagem Carrosinf.

Aproveitando o momento vou também deixar o código @RestController do nosso personagem USUARIO, aqui embaixo para entenderem:

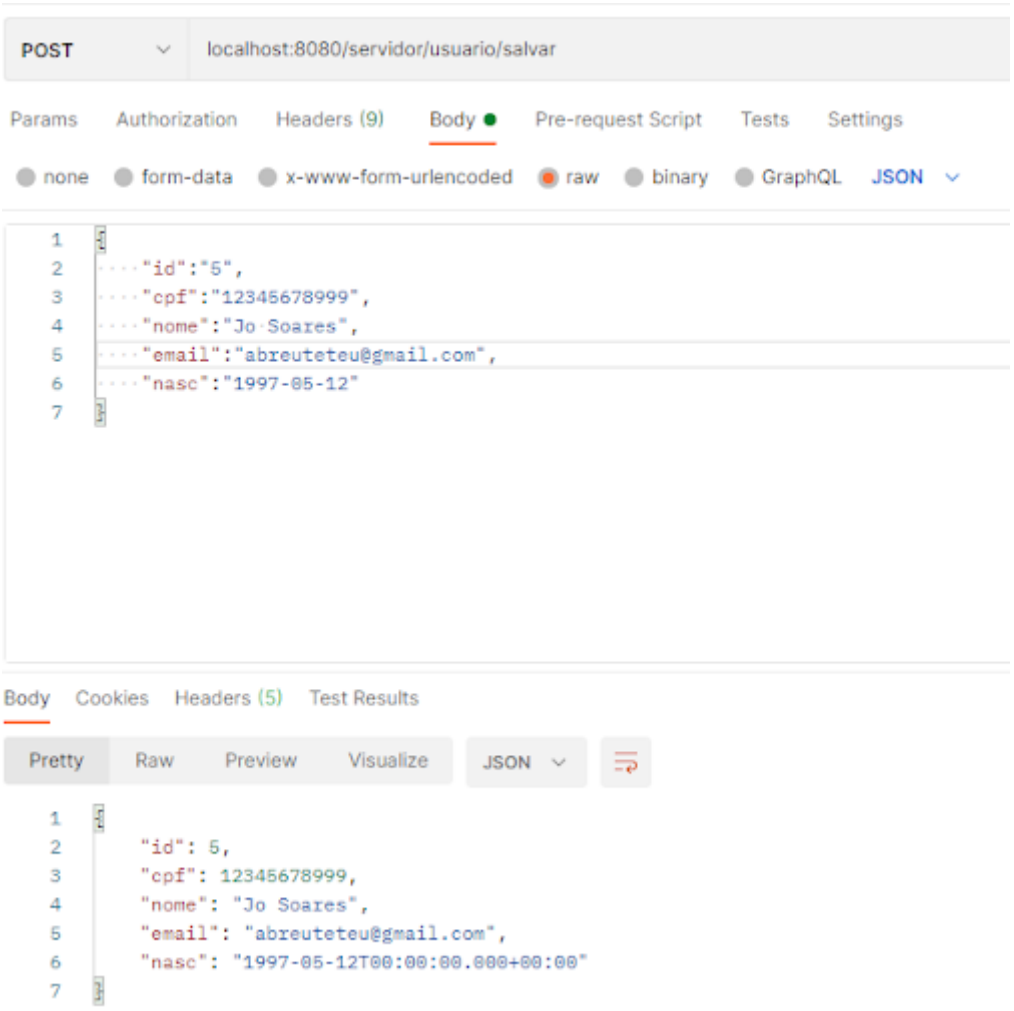
```
@RestController

 class                          Usuariostatus                          {

    @Autowired  //neste  momento  estamos  atribuindo  funcoees  ao
repositorio, se deixarmos sem o autowired o programa retornaria null
    private           Usuariorepositorio           usuariorepositorio;

    @GetMapping(path            =            "/servidor/usuario/{id}")
    public ResponseEntity consultar (@PathVariable("id") Integer id) {
        return                          usuariorepositorio.findById(id)
                .map(record                                          ->
ResponseEntity.status(HttpStatus.OK).body(record))
                .orElse(ResponseEntity.notFound().build());
    }
    @PostMapping(path           =           "/servidor/usuario/salvar")
    public   Usuarioval   salvar(@RequestBody   Usuarioval   usuario)   {
        return                          usuariorepositorio.save(usuario);
    }
}
```

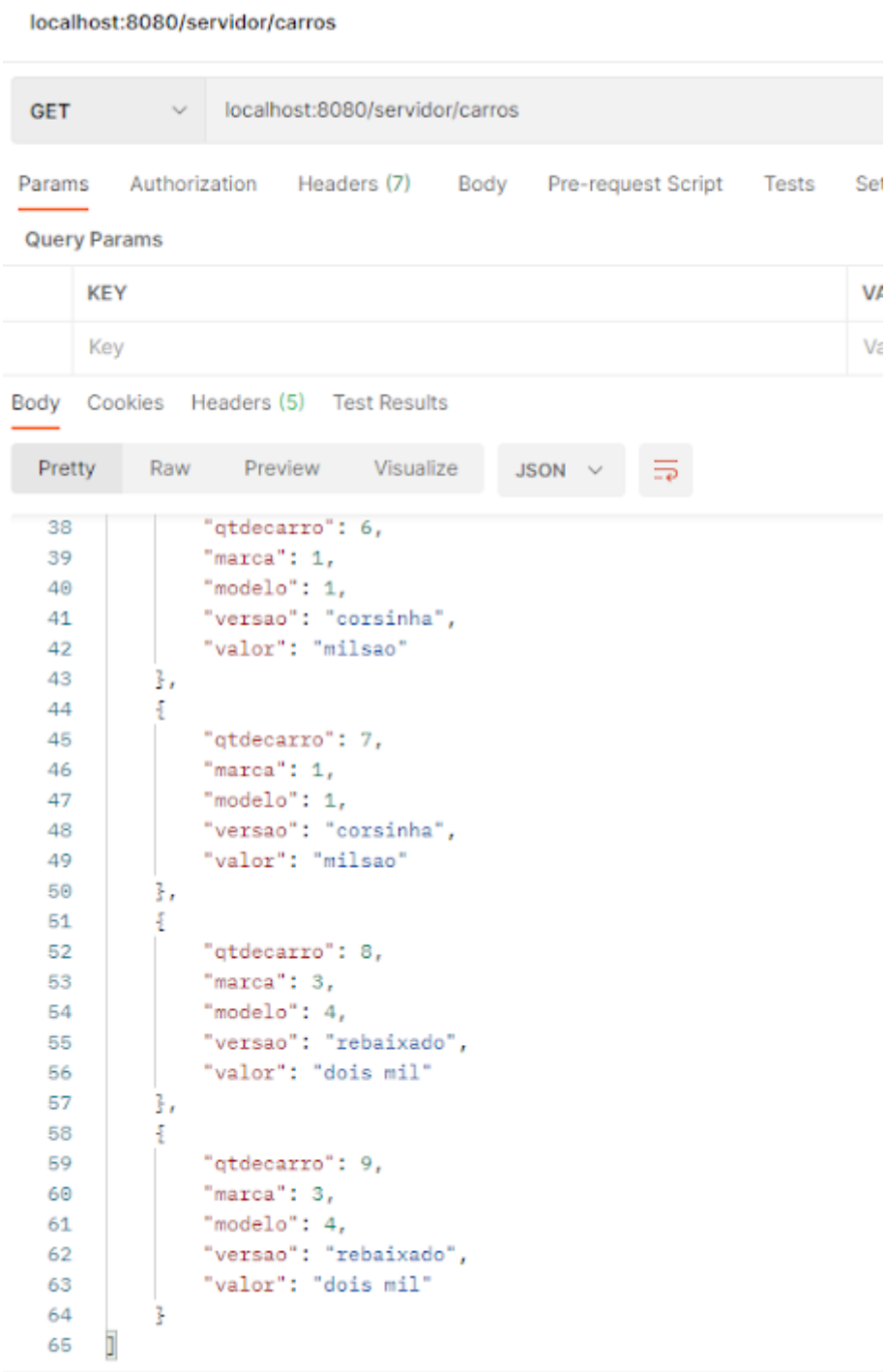
Obs: .map -- mostra o caminho e as acoes que devem ser feitas neste caminho.
.orElse -- mostra o que deve acontecer se ele nao achar nada neste caminho.
Usamos também o .findById(id), ou seja, toda vez que entrarmos no caminho `"/servidor/usuario/{id}"` encontraremos os dados do "id" / usuário especifico.

Agora que já temos nosso personagem, ambiente, ponte do arco-íris e um método de salvar, vamos fazer testes.

Hora dos Testes !!!

Vamos começar fazendo o teste de cadastro de usuário e sua consulta.


POST
localhost:8080/servidor/usuario/salvar
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"id":"5",
"cpf":"12345678999",
"nome":"Jo Soares",
"email":"abreuteteu@gmail.com",
"nasc":"1997-05-12"
}
Body
Cookies
Headers (5)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": 5,
"cpf": 12345678999,
"nome": "Jo Soares",
"email": "abreuteteu@gmail.com",
"nasc": "1997-05-12T00:00:00.000+00:00"
}



Hora de ver agora o teste do cadastro de nosso carros! Como podem ver ambos os testes foram um sucesso !

Infelizmente amigos, cheguei ate aqui com o tempo disponível que tinha. Mas como um grande guerreiro nunca larga sua espada em uma batalha, vou sugerir o que fazermos para terminamos o desafio.

E agora?

Amigos, vou ponderar o que temos ainda de fazer e como imagino que deveremos fazer estes últimos pontos ! Vamos
lá!

Primeiro vamos trocar a variável 'versão' que criamos para a variável ano, depois disto vamos criar um método que pegue o ultimo digito do ano informado e atribua ele a um dia da semana. Ex:
1 - Monday e assim por diante.
Depois criaremos um if, else para validarmos se o ultimo digito do ano do carro x bate com o dia da semana que deveria ser. Se sim , colocaríamos um return = true | else return = false.

Além disto teríamos que criar a tela de "boas vindas do usuário" informando qual carro possui seu rodizio ativo no dia.

Independente do prazo do Desafio ter acabo vou estar continuando ele como meta pessoal e atualizando vocês neste Blog.

Nos vemos em breve.