Criação de uma API REST de controle de veículos de usuários

Olá pequenos e grandes Devs!

Neste blog post vamos detalhar e evidenciar os passos para a criação de uma API REST que precisará controlar veículos de usuários. Procuraremos deixar tudo mais claro e simples possível para o nosso aprendizado e entendimento

E eu sei que assim como eu você também está ansioso, por isso, sem mais delongas, vamos colocar os traços nos t's, ou melhor, os elses nos ifs.

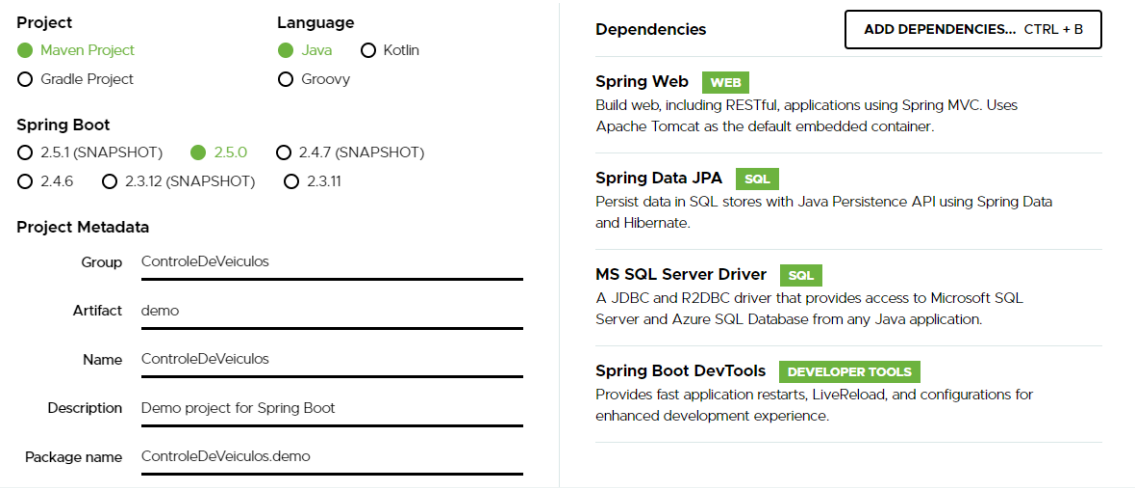
Para começar, vamos definir as nossas linguagens e ferramentas a serem utilizadas no nosso escopo. Tomamos como proposta o Java. Esta linguagem de programação é uma das que mais cresceu nos últimos anos e uma das mais utilizadas no mundo inteiro, mais do que uma linguagem, o Java é uma plataforma de desenvolvimento que se destaca pela sua praticidade, facilidade de integração com outras plataformas, diversas comunidades onde podem ser achadas e compartilhadas soluções e sem falar da JVM, para quem ainda não se decidiu entre o Windows, Linux ou MAC!

E acompanhado do Java tem o seu amigo Spring, o que eu chamo de "a pipoca que faltava na seção de cinema", "o pão de queijo que completa o café da tarde". O framework Spring vai facilitar o desenvolvimento da nossa aplicação. Temos com ele, a nossa disposição, recursos como segurança, integração, testes, desenvolvimento web, que é a proposta para a construção da nossa API e, também a possibilidade de criar soluções menos acopladas, mais coesas e fáceis de compreender e manter.

Vamos criar um projeto Maven para gerenciarmos as nossas dependências e controlarmos a versão dos nossos artefatos.

Para a nossa API vamos escolher algumas tecnologias Spring.

Poderemos acrescentar ao nosso escopo ao longo do nosso desenvolvimento.



Optamos pelo **JPA** para utilizarmos em nosso escopo o Spring Data e outros recursos do Java Persistence aplications e Hibernate, facilitando o mapeamento dos nossos atributos.

Traduzindo, poderemos converter os nossos objetos relacionais em tabelas no banco de dados.

Optamos também pelo Spring Web que nos ajudará a criar um web service, ou seja, vamos enviar e receber requisições através de protocolos, já que a nossa proposta é baseada em uma arquitetura REST. Toda essa questão da padronização de sistemas, a meu ver, facilita muito o processo de comunicação entre estes.

Usamos também o SQL Server Driver para termos acesso ao servidor SQL e programarmos o nosso banco de dados e o Devtools para melhorar o nosso desempenho enquanto trabalhamos no projeto.

Vamos começar por implementar a nossa classe de usuários:

```
public class Usuario {

    private String nome;
    private String email;
    private String cpf;
    private Data dataNascimento;
    //Gerar construtores
    // Getters e Setters
    // HashCode e Equals
}
```

Vamos adotar a implementação de um construtor para facilitar a instanciação dos nossos objetos e também deixar os nossos atributos privados para evidenciarmos o encapsulamento com a implementação dos nossos Getters e Setters. Vamos também gerar os Hashcode e equals, para que os nossos objetos sejam comparados pelo seu conteúdo e não pelo ponteiro de memória.

Em seguida, vamos implementar o nosso cadastro de veículos:

```
public class Veiculo {

    private String marca;
    private String modelo;
    private Integer ano;
    //Gerar construtores
    // Getters e Setters
    // HashCode e Equals
}
```

Criadas as classes, vamos programar a associação existente entre elas, pois, pressupõe-se que um usuário pode ter um carro ou mais de um carro.

```
Public class Usuario {

    private String nome;
    private String email;
    private String cpf;
```

```
    private Data dataNascimento;
    private List<Veiculo> veiculo = new ArrayList<>();
```

Para isso, dizemos que o usuário possui um atributo ‘veiculo’ que é do tipo ‘Veiculo’. No nosso caso teremos uma lista de ‘veiculos’.

Vamos também implementar em nossas classes a interface Serializable, uma interface que permite a conversão dos objetos das nossas classes em bytes para que estes possam ser gravados em arquivos e trafegar em redes.

```
public class Veiculo implements Serializable {...
public class Usuario implements Serializable {...
```

Em seguida, vamos fazer o mapeamento objeto relacional para que as nossas classes sejam geradas no banco de dados como tabelas. Para isso, podemos usar a anotação @Entity do jpa, que especifica que aquela classe será uma tabela do nosso banco de dados, desse jeito, os atributos pertencentes a classe serão colunas da mesma tabela.

Antes, vamos acrescentar um atributo id ao nosso cliente, para que sirva de identificador.

```
@Entity
public class Usuario implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    private String email;
    private String cpf;
    private Data dataNascimento;
```

Note que acrescentamos também algumas anotações ao id, desta forma informamos que este id é único e que os seus valores serão gerados e incrementados automaticamente pelo banco de dados para cada registro novo inserido.

Ainda podemos definir o relacionamento existente entre as duas entidades no banco de dados através de algumas anotações.

```
@Entity
public class Veiculo implements Serializable{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String marca;
    private String modelo;
    private Integer ano;

    @ManyToOne
    @JoinColumn(name="usuario_id")
    private Usuario usuario;
```

Com isso queremos dizer que um usuário pode ter vários veículos, através da anotação @ManyToOne e que o nome da coluna da chave estrangeira da tabela veiculo pertencente ao usuário vai se chamar “usuario_id”.

E do outro lado, mapeamos o usuário referenciando que o mesmo já foi mapeado dentro de veiculos pelo objeto usuário e que o seu relacionamento é de um para muitos, ou seja, um usuário pode ter vários veículos cadastrados em seu nome.

```
@Entity
public class Usuario implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nome;
    private String email;
    private String cpf;
    private Data dataNascimento;

    @OneToMany(mappedBy = "usuario")
    private List<Veiculo> veiculo = new ArrayList<>();
```

Todas essas declarações são feitas na nossa camada de domínio, onde comumente é apresentado o modelo relacional, muitas vezes representado por um diagrama de classe, tendo em conta as boas práticas de engenharia de software.

Um aspecto importante do nosso projeto é que alguns atributos foram definidos como obrigatórios. Podemos então usar uma anotação para validarmos isso. Vamos incluir também uma validação customizada dos nossos CPF e email para que estes sejam únicos através do pacote de dependência do Hibernate.validator ou Bean Validation.

```
@NotEmpty(message = "Preenchimento obrigatório")
private String nome;

@NotEmpty(message = "Preenchimento obrigatório")
@Email
private String email;

@NotEmpty(message = "Preenchimento obrigatório")
@CPF
private String cpf;

@NotEmpty(message = "Preenchimento obrigatório")
private Date dataNascimento;
```

Faremos o mesmo para os atributos de veículos, validando os campos obrigatórios.

```
@NotEmpty(message = "Preenchimento obrigatório")
private String marca;

@NotEmpty(message = "Preenchimento obrigatório")
private String modelo;
```

```
private Integer ano;
```

Note que não validamos o atributo ano, pois, o a anotação utilizada serve apenas para atributos strings. Podemos então pensar em converter esse campo em alguma etapa do processo.

Agora podemos instanciar alguns clientes e veículos, mas para que estes sejam salvos no banco de dados precisamos criar o nosso repository, que representa a camada de acesso a dados, muitas vezes também chamada de DAO, esta tem a função de conversar com o banco de dados.

```
import ControleDeVeiculos.demo.dominio.Usuario;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UsuarioRepository  extends JpaRepository<Usuario, Integer> {
}
```

Nesse caso, implementamos uma interface que vai herdar da interface JpaRepository capaz de acessar os dados com base nos objetos que passamos para ela e no tipo o atributo identificador, no caso, o Id.

Podemos então, criar o nosso endpoint para o nosso usuário, que vai estar localizado de Resources ou como é chamada, camada de controladores, que é responsável por controlar as nossas requisições REST.

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/usuarios")
public class UsuarioResource {
}
```

Para isso usamos a anotação @RestController que é uma diretiva do Spring boot e também a anotação @RequestMapping que define o caminho/endpoint que vamos usar para esse recurso, ou seja, se quisermos acessar os usuário da nossa aplicação no navegador ou no postman, vamos usar o “http//localhost:8080/usuario” por exemplo.

Para que o nosso método seja uma função REST precisamos associar ela com alguns dos verbos HTTP, pois é importante que sejam atribuídos os verbos adequados para cada operação desejada. Para o nosso caso, onde queremos obter dados nós usamos o GET, se quiséssemos apagar utilizaríamos o DELETE, ou o POST, se quiséssemos introduzir um novo dado e assim por diante.

```
@RestController
@RequestMapping(value = "/usuarios")
public class UsuarioResource {

    @RequestMapping(method = RequestMethod.GET)
    public Usuario find (Integer id){

        return Usuario;
    }
}
```

Porém, para que o nosso controlador seja capaz de acessar os nossos dados, será necessário a implementação de um serviço responsável por buscar o nosso usuário, este por sua vez chama o nosso repository que busca o nosso objeto no banco de dados.

```
@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repo;

    public Usuario buscar(){

    }
}
```

Para isso, vamos usar um método que busca os nossos usuários por Id, para isso chamamos uma operação do objeto de acesso a dados, declarando uma dependência do tipo “UsuarioRepository”, evidenciando uma das principais funcionalidades do Spring que é a injeção de dependências e, usamos a anotação “@Autowired” para instanciar a nossa dependência de forma automática dentro da classe.

```
@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository repo;

    public Usuario buscar(Integer id){
        Usuario obj = repo.findAll(id);
        return obj;
    }
}
```

Agora que temos a nossa busca implementada vamos fazer algumas alterações no nosso UsuarioResource e alterar o nosso endpoint:

```
@RestController
@RequestMapping(value = "/usuarios")
public class UsuarioResource {
```

```
    @Autowired
    private UsuarioService service;

    @RequestMapping(value= "/{id}", method = RequestMethod.GET)
    public ResponseEntity<?> find (@PathVariable Integer id){
        Usuario obj = service.buscar(id);

        return ResponseEntity.ok().body(obj);
    }
}
```

Acrescentamos o id ao nosso endpoint para mudar o parâmetro de busca, a partir desse ponto o nosso método de busca será /usuarios/id, sendo que este id é passado pela nossa anotação @PathVariable. O nosso método passa a retornar um tipo especial de resposta HTTP do serviço REST. Instanciamos também o service dentro da nossa classe para recebermos o método de busca que foi implementado no UsuarioService e fazemos retornar o nosso ResponseEntity com o nosso objeto.

Para obter os dados do valor dos nossos veículos vamos então consumir a API sugerida utilizando o Spring-Cloud-Feign. Utilizaremos este recurso por facilitar a integração de forma fácil usando apenas algumas linhas de código.

Para isso vamos adicionar duas dependências ao nosso pom. XML

Spring-cloud-starter-openfeign

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

spring-cloud-dependencies

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-dependencies</artifactId>
         <version>${spring-cloud.version}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

Em seguida, precisamos adicionar a anotação @EnableFeignClients à nossas classe principal para habilitar o Feign

```
@SpringBootApplication
@EnableFeignClients
public class ControleDeVeiculosApplication {
```

```
    public ControleDeVeiculosApplication() throws ParseException {
    }

    public static void main(String[] args) {
        SpringApplication.run(ControleDeVeiculosApplication.class, args);
    }

}
```

Com a nossa classe que retorna o nosso objeto veiculo já criada, criaremos uma interface onde o @FeignClient fará a chamada do serviço.

Os parâmetros Url e name serão obrigatórios, pois, um será a base do serviço que vai ser consumido e o outro o nome do usuario. Precisaremos também dizer qual o endpoint o buscarVeiculoPorAno vai utilizar.

```
@FeignClient(url= "https://deividfortuna.github.io/fipe/" , name =
"anoCarro")
public interface AnoVeiculoService {

    Veiculo buscarVeiculoPorAno(@PathVariable("ano") String ano);
}
```

Agora, criaremos a rota que usaremos para consumir o serviço.

```
@RestController
public class AnoCarroRestService {

    @Autowired
    private AnoVeiculoService anoVeiculoService;

    public ResponseEntity<Veiculo> getAno(@PathVariable String ano){
        Veiculo veiculo = anoVeiculoService.buscarVeiculoPorAno(ano);

        return veiculo !=null ? ResponseEntity.ok().body(veiculo) :
ResponseEntity.notFound().build();
    }
}
```

Pronto, agora a nossa aplicação está consumindo outra API externa.

Vamos fazer agora os nossos endpoints para cadastrarmos um novo usuário. O processo se assemelha a criação do método de busca que implementação anteriormente.

```
@Transactional
public Usuario insert (Usuario obj){
    obj.setId(null);
    obj = repo.save(obj);
    VeiculoRepository.saveAll(obj.getVeiculo());
    return obj;
}
```

Implementamos o método de busca no nosso UsuarioService e em seguida definimos o tipo de requisição REST que queremos para aquele método no nosso UsuarioResource.

```
@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<Void> insert(@Valid @RequestBody Usuario obj) {
    Usuario obj = service.insert(obj);
    obj = service.insert(obj);
    URI uri =
ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand
(obj.getId()).toUri();
    return ResponseEntity.created((uri)).build();
}
```

Criaremos também o endpoint para o Veiculo.

```

@Transactional
public Veiculo insert (Veiculo obj){
    obj.setId(null);
    obj = repo.(obj);
    VeiculoRepository.saveAll(obj.getUsuario());
    return obj;
}
```

```
@Autowired
private VeiculoService vservice;

@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<Void> insert(@Valid @RequestBody Veiculo obj) {
    Veiculo obj = vservice.insert(obj);
    obj = vservice.insert(obj);
    URI uri =
ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand
(obj.getId()).toUri();
    return ResponseEntity.created((uri)).build();
}
```

Criados todos os nossos endpoints, vamos implementar uma pequena lógica de rodízio

Vamos montar uma lógica na nossa classe veículo, declarando a variável DiadoRodizio.

Primeiramente, fazemos o dia de rodízio receber o ano do carro convertido em string, pegamos apenas os últimos dois dígitos e em seguida aplicamos a lógica.

```
public void getDiaDoRodizio(String diaDoRodizio, String d) {
    d =Integer.toString(getAno());
    diaDoRodizio = d.substring(d.length()-2);
    if(diaDoRodizio.equals(1) || diaDoRodizio.equals(0)){
        diaDoRodizio = "segunda-feira";
    } else if(diaDoRodizio.equals(2) || diaDoRodizio.equals(3)){
        diaDoRodizio = "terça-feira";
    }else if(diaDoRodizio.equals(4) || diaDoRodizio.equals(5)){
        diaDoRodizio = "quarta-feira";
    }else if(diaDoRodizio.equals(6) || diaDoRodizio.equals(7)){
```

```
        diaDoRodizio = "quinta-feira";
    } else if(diaDoRodizio.equals(8) || diaDoRodizio.equals(9)){
        diaDoRodizio = "sexta-feira";
    }
return DiaDoRodizio
```

Em seguida, fazemos o serviço correspondente retornar o dia do rodízio.

```
public String buscar(Integer id){
    Veiculo obj = repo.findAll(id);
    if  (obj == null) {
        throw new ObjectNotFoundException("Objeto não encontrado! Id: " + id
+ Usuario.class.getName());
    }
    return obj + obj.getDiaDoRodizio();
}
```

Vamos também criar um atributo de rodízio ativo e implementar a lógica para ele no nosso serviço.

```
public Serializable getRodizioAtivo(Calendar dataSemana, String  nome,
Integer dia) {

    switch(dia){
        case Calendar.SUNDAY: nome = "Domingo";
        break;
        case Calendar.MONDAY: nome = "segunda-feira";
        break;
        case Calendar.TUESDAY: nome = "terça-feira";
        break;
        case Calendar.WEDNESDAY: nome = "quarta-feira";
        break;
        case Calendar.THURSDAY: nome = "quinta-feira";
        break;
        case Calendar.FRIDAY: nome = "sexta-feira";
        break;
        case Calendar.SATURDAY: nome = "sábado";
        break; }
        if(nome == getDiaDoRodizio()){
            return true;
        }else {
            return false;
        }
}
```

Atribuimos os nomes aos dias do sistema iguais aos nomes da condicional anterior e então comparamos e em seguida fazermos o serviço retornar o seu resultado booleano.

```
@Transactional
public String buscar (Integer id){
    Veiculo obj = repo.findAll(id);
    if  (obj == null) {
        throw new ObjectNotFoundException("Objeto não encontrado! Id: " + id
+ Usuario.class.getName());
```

```
    }
    return obj + obj.getDiaDoRodizio() + obj.getRodizioAtivo();
}
```

Para fecharmos com chave de ouro, vamos personalizar alguns erros para que os mesmos fiquem padronizados e para que fiquem mais perceptíveis.

Podemos criar uma classe de padronização na nossa camada Resources.

```
public class StandardError implements Serializable {


    private Long timestamp;
    private Integer status;
    private String error;
    private String message;
    private String path;
   //Construtores
   //Getters e Setters


}
```

Criamos também uma classe de validação da nossa lista de erros que vai herdar da nossa classe StandardError

```
public class ValidationError extends StandardError {
    private java.util.List<FieldMessage> List = new ArrayList<>();

    public ValidationError(Long timestamp, Integer status, String error,
String message, String path) {
        super(timestamp, status, error, message, path);

    }

    public List<FieldMessage> getLista() {
        return List;
    }

    public void addError(String fieldName, String message){
        List.add(new FieldMessage(fieldName, message));
    }
```

Para instanciarmos um Field message vamos criar a classe que vai conter a nossa lista de campos e a mensagem personalizada.

```
public class FieldMessage implements Serializable {

    private String fieldName;
    private String message;
    //Construtores
   //Getters e Setters

    }
```

Vamos agora criar as exceções no nosso serviço

```
public class DataIntegrityException extends RuntimeException{
    public DataIntegrityException(String msg){
        super(msg);
    }
    public DataIntegrityException(String msg, Throwable cause)
    {
        super(msg, cause);
    }
}
```

```
public class ObjectNotFoundException extends RuntimeException{

    public ObjectNotFoundException(String msg)
    {

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

Por fim, criaremos o nosso manipulador de exceções na nossa camada de Resources.

```
@ExceptionHandler(ObjectNotFoundException.class)
public ResponseEntity<StandardError> objectNotFound(ObjectNotFoundException
e, HttpServletRequest request){

    StandardError err = new StandardError(System.currentTimeMillis(),
HttpStatus.NOT_FOUND.value(), "Não encontrado", e.getMessage(),
request.getRequestURI());
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(err);
}


@ExceptionHandler(DataIntegrityException.class)
public ResponseEntity<StandardError> dataIntegrity(DataIntegrityException e,
HttpServletRequest request){

    StandardError err = new StandardError(System.currentTimeMillis(),
HttpStatus.BAD_REQUEST.value(), "Integridade de dados", e.getMessage(),
request.getRequestURI());
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(err);
}
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<StandardError>
validation(MethodArgumentNotValidException e, HttpServletRequest request){

    ValidationError err = new ValidationError(System.currentTimeMillis(),
HttpStatus.UNPROCESSABLE_ENTITY.value(), "Erro de validação", e.getMessage(),
request.getRequestURI());
```

```
    for(FieldError x : e.getBindingResult().getFieldErrors()){
        err.addError(x.getField(), x.getDefaultMessage());
    }
    return ResponseEntity.status(HttpStatus.UNPROCESSABLE_ENTITY).body(err);
}
```

Pronto Dev! Terminamos de construir a nossa API REST!

Espero que eu tenha Ajudado e que apartir daqui você dar um passo a mais para se tornar um Dev ainda mais qualificado.

Deixaremos aqui descrito o link do repositório no github para acessar a implementação do projeto: https://github.com/Herlander929/Contrtole-De-Veiculos.

Por: Herlander Francisco Sanguluvulo da Silva.