Sumário

Regras de Negocio

Contruir uma API para controle de um estacionamento

Dados cadastrados em BD Relacional

Exposta na nuvem

Controle de acesso

RNF

Guardar placa cor e modelo

Guardar data e hora de entrada

Calcular valor para saída

Um tipo de usuário, o operador do estacionamento

Introdução

Spring Boot

Framework para facilitar processo de configuração e publicação dos Serviços Web

Rodar a aplicação mais rápido possível

Spring Framework

Criado para evitar a complexidade do Java EE, é um complemento do Java EE para facilitar nas requisições

Spring Dependencias

- Spring Data – Vincular ao Banco de dados
- Spring Cloud – Armazenar na nuvem
- Spring Security – Seguranca da aplicação
- Spring Web – API na Web

- Lombok – ajuda nos get e sets
- DevTools – Não é necessário rodar toda vez, quando salva a aplicação atualiza solo

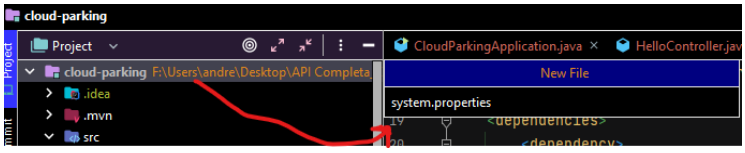
  E para atualizar solo → Build – Recompile...;

- H2 – banco de dados na nuvem

Implantando Codigo no Heroku

Heroku é uma aplicação que pega o código e compila, após compilado, disponibiliza um link web para poder ver a aplicação

Por default ele utiliza o Java 8, para mudar do padrão, implantar o código no projeto, dizendo que vc quer colocar para a versão, no meu caso “Java 17”;



Dentro desse File...:

```
java.runtime.version=17
```

Quando da erro no Heroku por não achar a linguagem

```
-----> Building on the Heroku-22 stack
-----> Determining which buildpack to use for this app
 !     No default language could be detected for this app.
                            HINT: This occurs when Heroku cannot detect the buildpack to
use for this application automatically.
                            See https://devcenter.heroku.com/articles/buildpacks
 !     Push failed
```

Provavelmente por o arquivo não estar logo de cara, como nesse meu:

Ele tem varias pastas p aí chegar no SRC ...

Deve especificar o caminho para o sistema...

... Não sei ainda como resolver :/

Rest

Rest são os padrões de requisições de HTTP – modelo de arquitetura para sistemas distribuídos

Há os níveis de Rest, no total 4, o que torna quando os 4, REST FULL (Já falado em arquivos anteriores)

Verbos HTTP

Get

Anotado com **@GetMapping** (No controller como BOAS PRATICAS)

Faz a requisição de PEGAR os dados do Banco, *GetAll* para todos os dados, ou com *@GetMapping ("/id")* para um ID em especifico

Post

Anotado com **@PostMapping** (Pacote Controller[...])

Postagem dos dados no banco, sempre em formato JSON*(mas isso nao vem ao caso agora)*

Put

Anotado com **@PutMapping("/id")** (Pacote Controller[...])

Atualização dos dados, sempre especificando na URL o objeto que quer atualizar/alterar

Delete

Anotado com **@DeleteMapping("/id")** (Pacote Controller[...])

Deletar os dados do banco, sempre especificando na URL o objeto que quer deletar

Path

-_-_-_- Não usei ainda

Quando quer alterar algum campo em especifico, informar **Campo e Valor**

Declarando Controller

Declarado com @RestController & @RequestMapping(“/***urlDeReferencia***”)

Foi declarado a Entidade Própria do nosso Banco de Dados, **isso não é uma boa pratica**

Por isso deve ser feito o DTO

Nunca se expõe o objeto de domínio na aplicação

Para isso, criar camada de Serviço, para definir o domínio

Criando Service

Declarado com @Service, injetando dependência Service vinculado ao controller

Quando implementa Service →

Inversão De Controle (IOC) - Service Em Controller

EM VEZ DE @AUTOWIRED ParkingService parkingService

```java
@RestController
@RequestMapping("/parking")
public class ParkingController {
//    Listar todos os estacionados nas vagas
    @GetMapping
    public List<Parking> findAll(){
        Parking parking = new Parking();
        parking.setColor("Preto");
        parking.setLicense("AAA-1111");
        parking.setModel("Audi A5");
        parking.setState("CWB");
        return Arrays.asList(parking, parking);
    }
}
```

COLOCAR

private final ParkingService parkingService

Deixando Codigo mais Limpo

Agora, em vez de feito como acima, declarando tudo aquilo em Controller

```java
uestMapping("/parking")
ic class ParkingController {

/*PRIVATE É DEFAULT, NAO NECESSARIO DECLARAR*/
private final ParkingService parkingService;

/*Construtor criado Automatico*/
public ParkingController(ParkingService parkingServi
    this.parkingService = parkingService;
}

//    Listar todos os estacionados nas vagas
@GetMapping
public List<Parking> findAll(){
    return parkingService.findAll();
}
```

Apagar o que feito em Controller

Declarar dessa forma em Service:

```java
@Service
public class ParkingService {

    private static Map<String, Parking> parkingMap = new HashMap<>();

    static {
        String id = getUUID();
        Parking parking = new Parking(id, license: "AAA-1112", state: "BR", model: "Focus", color: "Preto");
        parkingMap.put(id, parking);
    }

    public List<Parking>findAll(){
        /*LEMBRANDO Arrays.asList -> adiciona manualmente os itens dentro do construtor, por isso Cont
//        return Arrays.asList().add();
//        return parkingMap.values().stream().collect(Collectors.toList()); MELHORANDO
        return new ArrayList<>(parkingMap.values());
    }

    private static String getUUID() {
        return UUID.randomUUID().toString()/*Convert p String*/.replace(target: "-", replacement: " ");
    }
```

Como pode ver

Toda aquela poluição visual no controller, foi para o Service, onde ficam as regras de negócio


```
[{"id":"24b3de3f 94b9 4124 b2f5 f3433ad685ba","license":"AAA-
1112","state":"BR","model":"Focus","color":"Preto","entryDate":null,"exitDate":null,"bill":null}]
```


/*LEMBRANDO Arrays.asList -> adiciona manualmente os itens dentro do construtor, por isso Contrutor Vazio*/

UUID atualmente não está mais sendo feita na ***Entidade***, mas sim no ***Service*** como uma Regra de Negócio, **PEGANDO UUID → CLASSE DO JAVA E CONVERTENDO PARA STRING**

```java
private static String getUUID() {
    return UUID.randomUUID().toString()/*Convert p String*/.replace(target: "-", replacement: " ");
}
```

Transferindo o que declarado em (Entidade)Parking para (Data Transfer → DTO)ParkingDTO

BOAS PRATICAS ISSO!!!

Se fosse fazer essa transferência de Entidade Para DTO manual ficaria um código muito Extenso...

ENVOLVENDO **FOR EACH**,

Convertendo dentro de Controller os GetENTIDADE para SetDTO

```java
@GetMapping
public List<ParkingDTO> findAll() {
    List<ParkingDTO>
    List<Parking> parkingList = parkingService.findAll();
    for (Parking parking : parkingList) {
        ParkingDTO dto = new ParkingDTO();
        dto.setId(parking.getId());
    }
}
```

Entao utiliza-se da biblioteca Mapper

ModelMapper

*Essa biblioteca deve-se adicionar no POM.XML digitando **modelmapper, já aparece – org.modelmapper***

```
public class ParkingMapper { //responsável por fazer conversão em vez de converter lá dentro de cada classe
```

então o que deve fazer para converter, primeiro mostrando **Classe Controller**, e depois da classe criada **MODEL_MAPPER**

```
//    Listar todos os estacionados nas vagas
@GetMapping
public List<ParkingDTO> findAll(){ /*Como Agora Editado para ParkingDTO, reeditar o retorno*/
/       return parkingService.findAll();
    List<Parking>parkingList = parkingService.findAll();
    List<ParkingDTO> result = parkingMapper.toParkingDTOList(parkingList);
}
```

```
@Component
public class ParkingMapper {//Responsavel por fazer conversão em vez de converter la dentro de cada classe

    public static final ModelMapper MODEL_MAPPER = new ModelMapper();

    public ParkingDTO parkingDTO(Parking parking){
    /* Declarando essa classe, ele pega os atributos de parking, compara de ParkingDTO, se forem iguais ou pareci
        return MODEL_MAPPER.map(parking, ParkingDTO.class);
    }
```

```
public List<ParkingDTO> toParkingDTOList(List<Parking> parkingList) {
    return parkingList.stream().map(this::parkingDTO).collect(Collectors.toList());
}
```

Voltando para Controller

```
//    Listar todos os estacionados nas vagas
@GetMapping
public List<ParkingDTO> findAll(){ /*Como Agora Editado para ParkingDTO, reeditar o retorno*/
      return parkingService.findAll();
    List<Parking>parkingList = parkingService.findAll();
    List<ParkingDTO> result = parkingMapper.toParkingDTOList(parkingList);
    return result;
}
```

Dessa forma os códigos ficam bem mais empacotados, é necessário ter um conhecimento pois é uma complexidade maior de código, ter a determinação para ler documentações(uma coisa que eu nn tenho, mas vamo indo neh)

Como fazer para não aparecer os campos Null / Nulos


```
[{"id":"24b3de3f 94b9 4124 b2f5 f3433ad685ba","license":"AAA-
1112","state":"BR","model":"Focus","color":"Preto","entryDate":null,"exitDate":null,"bill":null}]
```

No DTO

Colocar a Annotation @Json

```
@JsonInclude(JsonInclude.Include.NON_NULL)
```

```
[{"id":"ab7b0efd 216f 4a2d 851a c1a09e4a6583","license":"AAA-
1112","state":"BR","model":"Focus","color":"Preto"}]
```

Dessa forma ficando bem mais apresentável, em os campos null

Implementando Get By Id através do ModelMapper

Quando implementado buscando por ID, ele não retorna mais uma lista, então o método no ModelMapper é o mesmo, no entanto o modo de busca é diferente

Em controller

As chamadas quase as mesmas:

```
@GetMapping("/{id}")
public ResponseEntity<ParkingDTO>findById(@PathVariable String id){
    Parking parkingList = parkingService.findById(id);
    ParkingDTO /*Retorna um ...*/ result = parkingMapper.toParkingDTO(parkingList);
    return ResponseEntity.ok(result);
}
```

Em Service

```
public Parking findById(String id) {
    return parkingMap.get(id);
}
```

Busca por ID localhost:8080/parking/cd549965 abbd 49ff b68a 6a468b313aad

```
{"id":"cd549965 abbd 49ff b68a
6a468b313aad","license":"AAA-
1112","state":"RS","model":"Focus","color":"Preto"}
```

Modelando Post

Responde ao DTO

Por receber informações, não tem mais o PathVariable, e sim um ParkingDTO, pois recebe um corpo de informações

Para receber essas informações em DTO e passar para a Entidade, declarar assim no Mapper:

```
public Parking convertForParking(ParkingDTO parkingDTO) {
    return MODEL_MAPPER.map(parkingDTO, Parking.class);
}
```

No Service

```
//    POST
public Parking create(Parking parkingCreate) {
    /*Setar ID*/
    String uuid = getUUID();
    parkingCreate.setId(uuid); /*Seta e pega o ID veiculo, automatico*/
    parkingCreate.setEntryDate(LocalDateTime.now()); /*lança qual a data de entrada do veiculo, automatico*/
    parkingMap.put(uuid, parkingCreate);
    return parkingCreate;
}
```

No Controller

```
public ResponseEntity<ParkingDTO>create(@RequestBody ParkingDTO parkingDTO){
//      Para fazer essa criação, quero converter um DTO em um Parking, para isso, ou fazer a classe Convert,
    Parking parkingCreate = parkingMapper.convertForParking(parkingDTO); //Converter para DTO
    Parking parkingPost = parkingService.create(parkingCreate);
    ParkingDTO result = parkingMapper.convertForParkingDTO(parkingPost); //Converter para Parking
//      return ResponseEntity.ok(result);
//      return ResponseEntity.status(HttpStatus.OK).body(result);
    return ResponseEntity.status(HttpStatus.CREATED).body(result);
}
```

Agora, para fazer com que os parâmetros de envio sejam padrão , deve criar uma classe que so recebe esses parâmetros ➔ Converte ele em DTO ➔ Em Seguida em Service

MUITO CONFUSO SLK

No Controller Com Classe Requisições (ATUALIZADO)

```
@PostMapping
public ResponseEntity<ParkingDTO>create(@RequestBody ParkingRequisicoesDTO parkingRequisicoesDTO){ /*Con
/       Para fazer essa criação, quero converter um DTO em um Parking, para isso, ou fazer a classe Conver
    Parking parkingCreate = parkingMapper.parkingCreateDTO(parkingRequisicoesDTO); //Converter para DTO
    Parking parkingPost = parkingService.create(parkingCreate);
    ParkingDTO result = parkingMapper.convertForParkingDTO(parkingPost); //Converter para Parking
      return ResponseEntity.ok(result);
/     return ResponseEntity.status(HttpStatus.OK).body(result);
    return ResponseEntity.status(HttpStatus.CREATED).body(result);
}
```

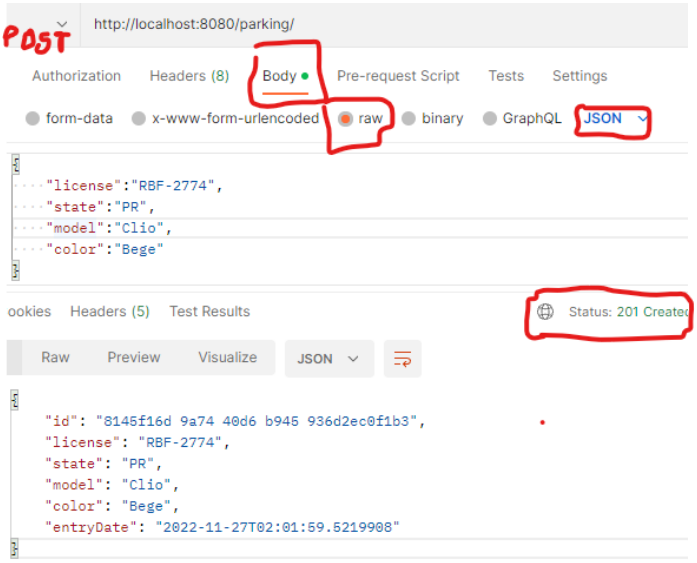
No Service Com Classe Requisições

Ele permanece o mesmo, sem alterações

No ModelMapper Com Classe Requisições (ATUALIZADO)

```
public Parking parkingCreateDTO(ParkingRequisicoesDTO
parkingRequisicoesDTO) {
    return MODEL_MAPPER.map(parkingRequisicoesDTO, Parking.class);
}
```

Sucesso na postagem



Modelando Delete

Depois de feito as outras requisições,

Com DELETE não tem segredo

Só fazer como fez para GET ONE / FOR ID:

No Controller

```
    @DeleteMapping("/{id}")
//    HAVIA COLOCADO COM @RequestParam, com esse Annotation ele pede esse parametro no BODY FORM-DATA
    public ResponseEntity<ParkingDTO> deleteID(@PathVariable String id){
        Parking parkingDelete = parkingService.deleteId(id);
        ParkingDTO result = parkingMapper.convertForParkingDTO(parkingDelete);
        return ResponseEntity.status(HttpStatus.ACCEPTED).body(result);
    }
```

Corrigindo na aula

```
//    HAVIA COLOCADO COM @RequestParam, com esse Annotation ele pede esse parametro no BODY FORM-DATA
//    Nao tem por que deixar o Response Entity com parking DTO pois ele vai retornar um objeto deletado = obj vazi
//
//    public ResponseEntity/*<ParkingDTO>*/ deleteID(@PathVariable String id){
//        Parking parkingDelete = parkingService.deleteId(id);
//        ParkingDTO result = parkingMapper.convertForParkingDTO(parkingDelete);
//        return ResponseEntity.status(HttpStatus.ACCEPTED).body(result);
//    }

    //REFAZENDO
    @DeleteMapping("/{id}")
    @ApiOperation("Delete Parking")
    public ResponseEntity deleteID(@PathVariable String id){
        parkingService.deleteId(id);
        return ResponseEntity.noContent().build();
```

No Service (NOTA: Classe para remover arquivos é o Remove, e não delete, então...)

```
public Parking deleteId(String id) {
    return parkingMap.remove(id);
}
```

Corrigindo na aula

```
public void deleteId(String id){
    Parking parkingDelete = findById(id);
    parkingMap.remove(id);
}
```

Modelando Put – Atualização

O Metodo é praticamente o mesmo que Post, mas com algumas alterações

Atualizar o Id existente, senão → Tratamento de exceções (mais abaixo)

Swagger

Depois de tudo Ok as requisições primarias

Adicionar o **SWAGGER** através do **Spring Fox**

- Ir no POM.XML
  - Digitar <dependência – quando digitado, ele se auto-completa
  - Digitar springfox-swagger2 - quando digitado, ele se auto-completa
  - Selecionar o relacionado ao SpringFox – quando aparecer a opção de groupID
- Fazendo isso e atualizando o POM, RODE O PROGRAMA
  - Procure o localhost:8080/Swagger.ui.html
- No entanto, quando se faz sem os códigos JSON, ele não tem como configurar, dessa forma...

Classe Config para Swagger

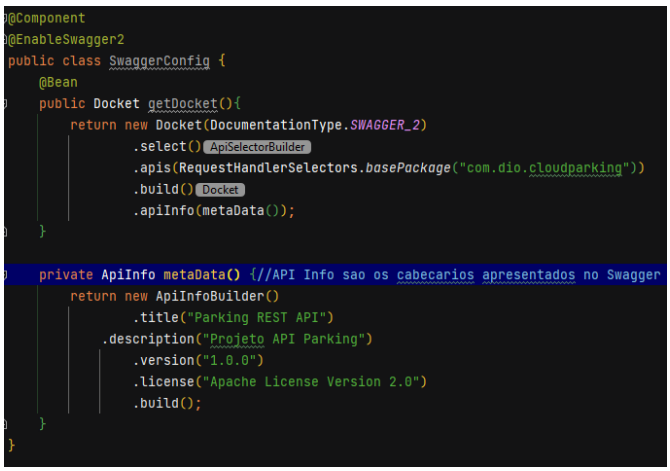
[...]Dessa forma, quando iniciar ele em vez de dar erro, já começa a aparecer os métodos implantados:

```
@Component
@EnableSwagger2
public class SwaggerConfig {
}
```

Quando implementado, pega todas as classes de requisições

Mas so quero que apareca as Requisições de Parking

Para isso...



```
@Component
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket getDocket(){
        return new Docket(DocumentationType.SWAGGER_2)
                .select() ApiSelectorBuilder
                .apis(RequestHandlerSelectors.basePackage("com.dio.cloudparking"))
                .build() Docket
                .apiInfo(metaData());
    }

    private ApiInfo metaData() {//API Info sao os cabecarios apresentados no Swagger
        return new ApiInfoBuilder()
                .title("Parking REST API")
            .description("Projeto API Parking")
                .version("1.0.0")
                .license("Apache License Version 2.0")
                .build();
    }
}
```

Configurando Swagger nos pacotes

@ApiIgnore → Quando quero ignorar aquela classe

Lembrando que se quiser ignorar outra classe que nn tenha nada haver com o projeto, assim como as requisições padrões do SpringFox Swagger

Colocar @Api Ignore

@Api → Quando quero que aquela classe apareca

São o pacote onde as requisições do controller estão,

É possível renomear elas com @Api(“API CONTROLLER PARKING”)

@ApiOperation → Quando quero renomear alguma requisição

As requisições, como GetOne GetAll GetNome... ou as demais operações

Para mapea-las com mais facilidade, declarar em cima do método com

@ApiOperation(“Get All IDs”) → Aparecera essa mensagem nessa requisição

Mas e se Caso ocorrer Algum Erro... Como tratar?

Há formas de tratamento de Excessoes

Todas através da SUPER CLASSE EXCEPTIONS, que extende para RUNTIMEEXCEPTIONS ou Demais

Erro de NotFound(404) apresentando como Internal Server Error(500) Como resolver?

Nesse projeto criaremos uma classe que Extende de RunTimeExceptions

Quando dá um erro de solicitação incorreta

Como no exemplo, o ID. Trataremos criando uma classe nova, dela estendendo RunTimeExceptions, com a Annotation para retornar um status 404 de não encontrada...:

```
//    GET ONE
public Parking findById(String id) { /*Fazer tratativa de erros*/
      return parkingMap.get(id);
    Parking parking = parkingMap.get(id);
    if (parking == null){
        throw new ParkingNotFoundException(id);
    }
    return parking;
}
```

ParkingNotFoundException

Fazendo isso, buscando a classe ParkingNotFoundException

```
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class ParkingNotFoundException extends RuntimeException{
    public ParkingNotFoundException(String id){
        super("Parking not found with Id: " + id);
    }
```

Fazendo essa Annotation

Ele capta quando ocorrer o erro, e visualiza o que é para dar retorno em vez de **500**

Aparecendo um erro muito extenso, como deixar melhor a exibição para Usuário ou FrontEnd (tratar erro nos Recursos)

Quando aparece um erro enorme, deixa muita poluição visual, acabando deixando o usuário se perder no próprio erro

Para tratar isso, deve ir no pacote recursos e deixar dessa forma as configurações

```
server.error.include-exception=false
server.error.include-stracktrace=never
server.error.include-message=always
```

Deixa a aplicação e aparição dos erros mais simples e prático entendimento

Preparando Persistência no Banco (JPA, HIBERNATE)

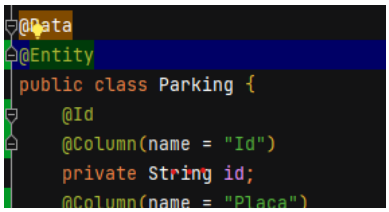
Colocar Dependencia do Data JPA com Driver Postgress

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

Criando a Annotation @Entity

Quando cria uma entidade, o primeiro passo é criar a anotação mostrando que se trata de uma entidade

E com isso, ele pede um ID que sera a chave primaria da classe (COISA DE BANCO DE DADOS)



Configurações de Persistência no Banco em “Resources”

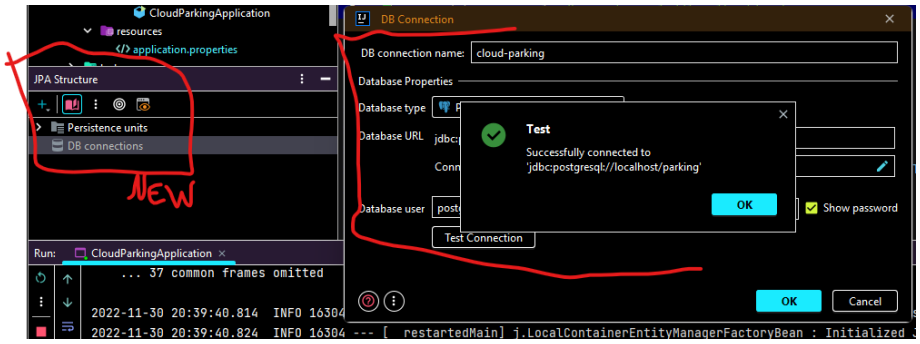
Mesmo fazendo isso ele apresenta erro quando “Run” no projeto

Para isso em Recursos...

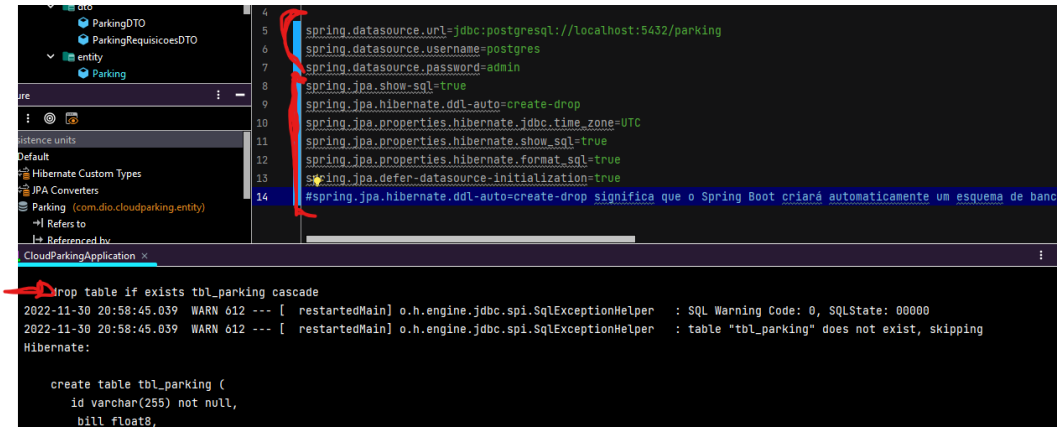
Deu Errooooooo, mas já resolvido

Deu erro: ***org.hibernate.tool.schema.spi.CommandAcceptanceException: Error executing DDL "create table [...]***

Descobrir mais p frente como arrumar, vai estar informado no Commit quando descobrir (:/)



Deu certo, testando conexão, incluindo alguns milhões de códigos, que não sei se alguns deles foram necessários, mas agora deu UHUUULskskskk



```
spring.datasource.url=jdbc:postgresql://localhost:5432/parking
spring.datasource.username=postgres
spring.datasource.password=admin
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.jdbc.time_zone=UTC
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.defer-datasource-initialization=true
#
```

O começo não se aplica ao projeto pois tem haver com o BD Azure

Mas em questão de linkar o JPA e DLL é isso...

https://learn.microsoft.com/pt-br/azure/developer/java/spring-framework/configure-spring-data-jpa-with-azure-sql-server

Após implementação Banco, Preparando camadas para requisições no banco!

Após feito todo o preparatório do banco

As requisições devem ser feitas através do banco e persistir nele

Para isso, Implementar o Service com JPA e o Repositório finalmente começa a funcionar

Repositório finalmente começa a funcionar

```
@Repository
public interface ParkingRepository extends JpaRepository<Parking,
String> { }
```

Implementar o Service com JPA

Após criado, vamos parar de utilizar a classe Map para as requisições

Dessa forma utilizar da classe, ou melhor, **interface Repository**

(*LOGO AGR QUE COMECEI A APRENDER SOBRE @AUTOWIRED ELE NÃO É MAIS UMA BOA PRÁTICA ~Stonks*)

Então... private final ParkingRepository parkingRepo;

E criar seu construtor

Implementar ele em vez do map, dessa forma

```java
    return parkingRepo.findAll();
}

//    Convertendo a forma de aparecer o ID de 111-111-111 para 111 111 111
private static String getUUID() { return UUID.randomUUID().toString()/*Convert p String*/.replace( target: "-",

//    GET ONE
public Parking findById(String id) { /*Fazer tratativa de erros*/
      return parkingMap.get(id);
      Parking parking = parkingMap.get(id);
    if (parking == null){
        throw new ParkingNotFoundException(id);
    }
    return parking;*/

    return parkingRepo.findById(id).orElseThrow(() ->
            new ParkingNotFoundException(id));
}

//    POST
public Parking create(Parking parkingCreate) {
    /*Setar ID*/
    String uuid = getUUID();
    parkingCreate.setId(uuid); /*Seta e pega o ID veiculo, automatico*/
    parkingCreate.setEntryDate(LocalDateTime.now()); /*lança qual a data de entrada do veiculo, automatico*/
    parkingRepo.save(parkingCreate);
    return parkingCreate;
```

Como a interface extende do JPA Repository,

O JPA já tem os métodos simples de um crud,

Dessa forma ele pode ser bem compactado

@Transactional

No caso, ele serve caso estoure algum erro

Ele faz a tratativa de erro caso desse erro e não salvasse e desse algum problema, ele faria o rolback

Tem seus tipos de transação

```java
//    GET ONE
@Transactional(readOnly = true)
public Parking findById(String id) { /*Fazer tratativa de erros*/
      return parkingMap.get(id);
      Parking parking = parkingMap.get(id);
    if (parking == null){
        throw new ParkingNotFoundException(id);
    }
    return parking;*/

    return parkingRepo.findById(id).orElseThrow(() ->
            new ParkingNotFoundException(id));
}

//    POST
@Transactional
public Parking create(Parking parkingCreate) {
    /*Setar ID*/
    String uuid = getUUID();
    parkingCreate.setId(uuid); /*Seta e pega o ID veiculo, automatico*/
    parkingCreate.setEntryDate(LocalDateTime.now()); /*lança qual a data de entrada do veiculo, automatic
    parkingRepo.save(parkingCreate);
    return parkingCreate;
```

Aplicando Spring Security (Salvando em memoria)

Lembrando que colocando dependência

```
spring-boot-starter-security
```

Ele já libera uma chave de acesso para entrar na aplicação

Using generated security password: 7b287efa-c7b0-47ea-807f-107ef995375d

@EnableWebSecurity

Criar através do config

Uma nova classe, onde gera tokens para os usuários, ou de forma mais fácil,

Criar um usuário e senha em memoria

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
//    Configurar login usuario
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
//        super.configure(auth);
        auth.inMemoryAuthentication() InMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
                .withUser( username: "Andre") UserDetailsManagerConfigurer<...>.UserDetailsBuilder
                .password(passwordEncoder().encode( rawPassword: "1234567890")) /*recebe a senha
                .roles("ADMIN")
                .and() InMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
                .passwordEncoder(passwordEncoder());
    }

    private PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```