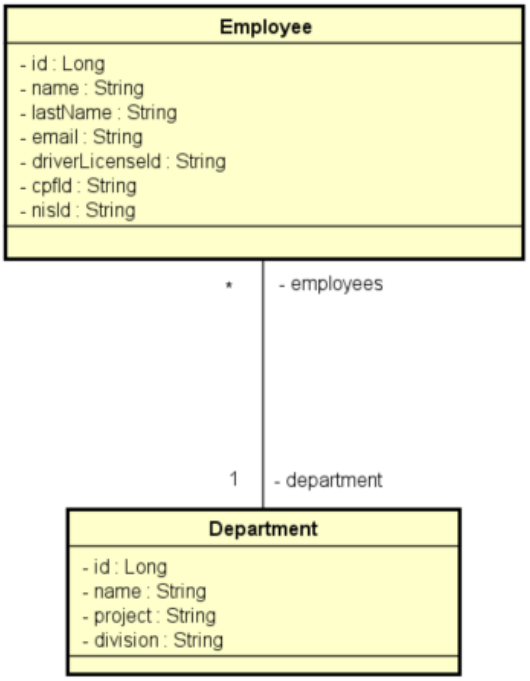
ARTIGO - O USO DA BIBLIOTECA MODELMAPPER

1.Introdução

Este artigo tem como objetivo mostrar alguns casos de uso da biblioteca ***ModelMapper*** (a biblioteca deve ser adicionada no arquivo POM.xml do projeto Spring Boot.), ela automatiza o mapeamento objeto ⟷ objeto evitando todo aquele código *boilerplate* quando precisamos trafegar informação de nossas entidades utilizando o padrão DTO.
O padrão DTO encapsula e serializa informação armazenada pelas entidades que é efetivamente requisitada, evitando o tráfego desnecessário de informação pela rede, além de desacoplar o *model layer* (entidades) da camada de apresentação (*controller layer*).

O modelo conceitual abaixo é utilizado para exemplificar o uso do *modelmapper*, onde existem duas entidades (empregado e departamento) onde a entidade departamento apresenta uma relação 1—>N com a entidade empregado, ao passo que o empregado só pode pertencer a 1 único departamento.



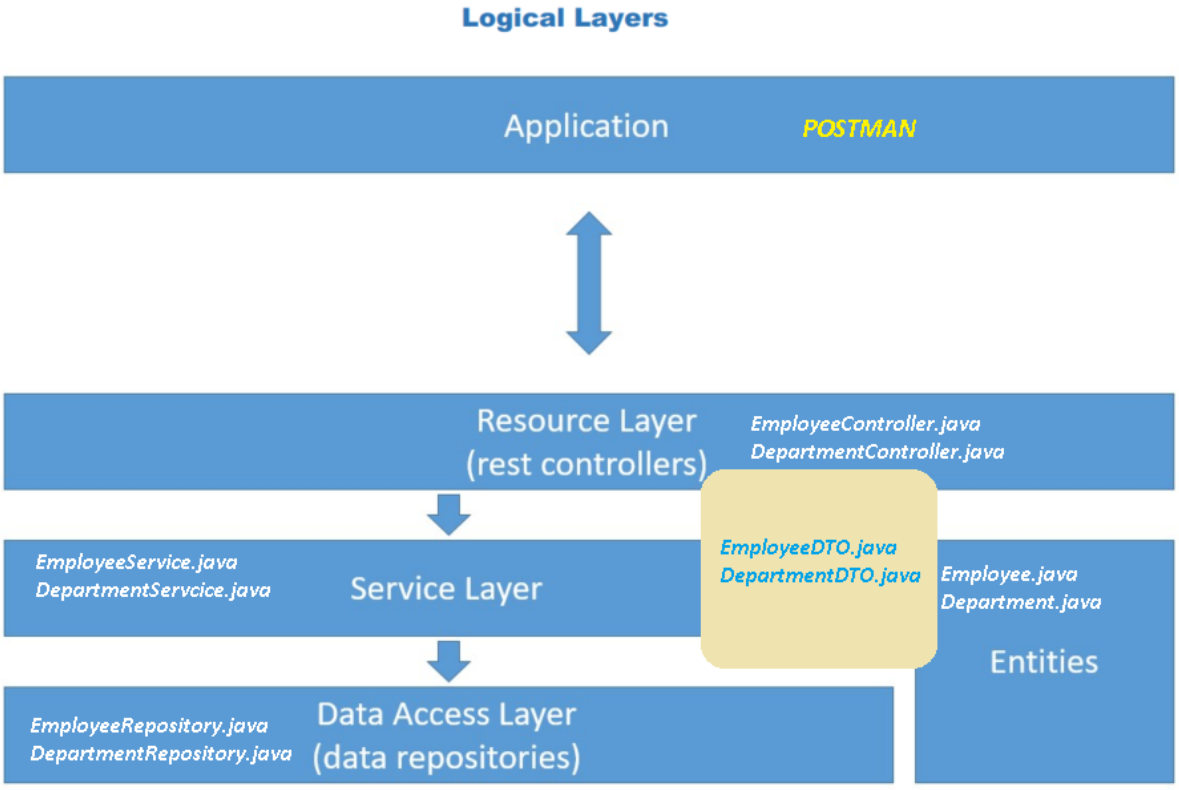
Queremos construir um *endpoint* que busque uma lista paginada de empregados, mas não queremos obter todos os atributos da entidade empregado, apenas algumas informações como id, nome e email, da entidade departamento igualmente, não queremos todos os atributos, apenas, nome e id do departamento. As camadas lógicas, usando a API REST, serão construídas da seguinte forma:

Onde *resource layer* será responsavel por interpretar as requisições do *frontend* forcendo as respostas adequadas no formado JSON.

O *service layer será responsável* mapeamento DTO⟷Entidade e tratará qualquer tipo de regra de negócio que porventura seja um requisito do modelo, em nosso caso estamos apenas interessados no mapeamento que será aplicado pelo *modelmapper.*

Finalmente o *repository layer* processará o acesso ao banco de dados onde estão armazenadas as informações da entidades relacionadas.

A aplicação será emulado através do aplicativo *POSTMAN*



2.Definindo as entidades Employee e Department

As duas entidades *Employee* e *Department* são criadas através de 2 classes POJO com os respectivos *@annotations* para a criação dos relacionamentos entre as tabelas do banco de dados.

```
//*************************
// classe Employee
//*************************
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "tb_employee")
```

```
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String name;
    public String lastName;
    public String email;
    public String driverLicenseId;
    public String cpfId;
    public String nisId;



    @ManyToOne
    @JoinColumn(name = "department_id")
    public Department department;

    public Employee() {
    }

    public Employee(Long id, String name, String lastname, St
            String nisId, Department department) {
        super();
        this.id = id;
        this.name = name;
        this.lastName = lastname;
        this.email = email;
        this.driverLicenseId = driverLicenseId;
        this.cpfId = cpfId;
        this.nisId = nisId;
        this.department = department;
    }


    public String getLastname() {
        return lastName;
    }
```

```
    public void setLastname(String lastname) {
        this.lastName = lastname;
    }

    public String getDriverLicenseId() {
        return driverLicenseId;
    }


    public void setDriverLicenseId(String driverLicenseId) {
        this.driverLicenseId = driverLicenseId;
    }


    public String getCpfId() {
        return cpfId;
    }


    public void setCpfId(String cpfId) {
        this.cpfId = cpfId;
    }

    public String getNisId() {
        return nisId;
    }


    public void setNisId(String nisId) {
        this.nisId = nisId;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
```

```
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

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

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }
}

//*************************
// classe Department
//*************************
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "tb_department")
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String name;
    public String division;
    public String project;

    @OneToMany(mappedBy = "department")
    public List<Employee> employees = new ArrayList<>();

    public Department() {
    }


    public Department(Long id, String name, String division, S
        super();
        this.id = id;
        this.name = name;
        this.division = division;
        this.project = project;
    }


    public String getDivision() {
        return division;
    }


    public void setDivision(String division) {
        this.division = division;
    }
```

```
    public String getProject() {
        return project;
    }


    public void setProject(String project) {
        this.project = project;
    }


    public Long getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Employee> getEmployees() {
        return employees;
    }
}
```

3.As classes DTO

Criamos então duas classes DTO com os atributos necessários para construir a resposta às requisições do *endpoint*:

```
//*************************
// classe EmployeeDTO
//*************************
import java.io.Serializable;

import com.rgiovann.modelmapper.usecase.entities.Employee;

public class EmployeeDTO implements Serializable {
    private static final long serialVersionUID = 1L;

    private Long id;
    private String name;
    private String email;
    private DepartmentDTO department;

    public EmployeeDTO() {
    }

    public EmployeeDTO(Long id, String name, String email, De
        this.id = id;
        this.name = name;
        this.email = email;
        this.department = department;
    }

    public EmployeeDTO(Employee entity) {
        id = entity.getId();
        name = entity.getName();
        email = entity.getEmail();
        department  = new DepartmentDTO(entity.getDepartment(
    }

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

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

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

    public DepartmentDTO getDepartment() {
        return department;
    }

    public void setDepartment(DepartmentDTO department) {
        this.department = department;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 : id.hash
        return result;
    }

    @Override
    public boolean equals(Object obj) {
```

```
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        EmployeeDTO other = (EmployeeDTO) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

}

//*************************
// classe DepartmentDTO
//*************************

import java.io.Serializable;

import com.rgiovann.modelmapper.usecase.entities.Department;

public class DepartmentDTO implements Serializable {
    private static final long serialVersionUID = 1L;

    public Long id;
    public String name;

    public DepartmentDTO() {
    }

    public DepartmentDTO(Long id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
public DepartmentDTO(Department entity) {
    id = entity.getId();
    name = entity.getName();
}

public Long getId() {
    return id;
}

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

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hash
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
```

```
        DepartmentDTO other = (DepartmentDTO) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

}
```

Veja os métodos em destaque nas duas classes DTO, toda vez que tenho que mapear uma entidade em uma classe DTO e vice-versa eu preciso fazer campo a campo, e se houver mudança nos atributos no futuro será necessário também modificar esses métodos, um acoplamento indesejado. A lib *modellmapper* realiza esse mapeamento automático. Vejamos como funciona.

4.A camada *Repository*

As classes do layer *Repository* são simples extensões da interface *JpaRepository* que fornece os métodos para acesso o bando de dados.

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.rgiovann.modelmapper.usecase.entities.Department;

public interface DepartmentRepository extends JpaRepository<D

}

import org.springframework.data.jpa.repository.JpaRepository;

import com.rgiovann.modelmapper.usecase.entities.Employee;

public interface EmployeeRepository extends JpaRepository<Emp
```

```
}
```

5.A camada *Controller*

Os *endpoints* que iremos criar são os seguintes:

- Uma lista paginada de todos os empregados inclusive seu departamento;
- Atualizar o nome ou email do empregado;
- Incluir um novo departamento (apenas o nome);

Para criar os *endpoints na camada* controller utilizamos os *@annotations do* Spring Boot, perceba que esta camada não enxerga as classes das entidades, apenas os seus DTOs:

```
//*****************************
// classe EmployeeController
//*****************************
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

import com.rgiovann.modelmapper.usecase.dto.EmployeeDTO;
import com.rgiovann.modelmapper.usecase.services.EmployeeServ

@RestController
@RequestMapping(value = "/employees")
public class EmployeeController {

    @Autowired
```

```
    private EmployeeService employeeService;

  // HTTP GET (lista comum)
    @GetMapping
    public ResponseEntity<Page<EmployeeDTO>> findAll(Pageable
        Page<EmployeeDTO> list = employeeService.findAllPaged
        return ResponseEntity.ok().body(list);
    }

  // HTTP PUT
    @PutMapping(value = "/{id}")
    public ResponseEntity<EmployeeDTO> update( @PathVariable
        productDTO = employeeService.update(id, productDTO);
        return ResponseEntity.ok().body(productDTO);

    }
}

//*****************************
// classe DepartmentController
//*****************************
import java.net.URI;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController
import org.springframework.web.servlet.support.ServletUriComp

import com.rgiovann.modelmapper.usecase.dto.DepartmentDTO;
import com.rgiovann.modelmapper.usecase.services.DepartmentSe

@RestController
@RequestMapping(value = "/departments")
```

```
public class DepartmentController {

    @Autowired
    private DepartmentService departmentService;

  // HTTP GET (paged)
    @GetMapping
    public ResponseEntity<List<DepartmentDTO>> findAll() {
        List<DepartmentDTO> list = departmentService.findAll(
        return ResponseEntity.ok().body(list);
    }

  // HTTP POST
    @PostMapping
    public ResponseEntity<DepartmentDTO> insert(@RequestBody
        departmentDTO = departmentService.insert(departmentDT
        URI uri = ServletUriComponentsBuilder.fromCurrentRequ
                .buildAndExpand(departmentDTO.getId()).toUri(
        return ResponseEntity.created(uri).body(departmentDTO

    }
}
```

6.Utilizando o *ModellMapper* na camada Service

Na camada de Serviço é que realmente utilizamos a biblioteca *modelmapper* para fazer o de-para entidade-DTO. Para a entidade Employee nós temos:

```
import java.util.NoSuchElementException;
import java.util.Optional;

import org.modelmapper.Conditions;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired
```

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transaction

import com.rgiovann.modelmapper.usecase.dto.EmployeeDTO;
import com.rgiovann.modelmapper.usecase.entities.Employee;
import com.rgiovann.modelmapper.usecase.repositories.Employee

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private ModelMapper employeeModelMapper;


    @Transactional(readOnly = true)
    public  Page<EmployeeDTO> findAllPaged(Pageable pageReque
            Page<Employee> page = employeeRepository.findAll(
            return page.map(p -> {EmployeeDTO employeeDTO = e
                                                return employee
                            }
                        );

    }


    @Transactional
    public EmployeeDTO update(Long id, EmployeeDTO employeeDT

            Optional<Employee> obj = employeeRepository.findB
            Employee entity = obj.orElseThrow(() -> new NoSuc

            employeeModelMapper.getConfiguration().setPropert
            employeeModelMapper.map(employeeDTO, entity); //
```

```
            employeeModelMapper.map(entity, employeeDTO); //
            return employeeDTO;

    }

}
```

Na busca paginada populamos uma nova instância da classe EmployeeDTO com os dados obtidos do retorno do método findAll (A), note que usamos o método map() + função lambda para transformar cada objeto Employee em EmployeeDTO (a classe page já é um *stream*)

No caso do *update* a forma de utilizar o mapper é diferente, perceba que o retorno da função é *void* e não um novo objeto, onde os parâmetros são o objeto fonte e o bjeto destino (C) e (D), a lib modellmapper permite várias customizações, uma delas é definir um condição onde o objeto destino só tem atributos atualizados se os atributos do objeto fonte são não nulos (B), dessa forma mesmo eu querendo atualizar apenas uma parte dos atributos do DTO, ele não atualiza os que estão nulos.

Perceba que não usei o método “save” pois como o a entidade está sob a supervisão do *Hibernate EntityManager* do Spring Boot, alterações nos atributos da entidade são espelhados no banco de dados, por isso em (C) eu carreguei novamente os valores que foram atualizados no banco de dados para o objeto DTO. No caso da entidade Departament, temos o seguinte:

```
import java.util.List;
import java.util.stream.Collectors;

import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transaction

import com.rgiovann.modelmapper.usecase.dto.DepartmentDTO;
```

```
import com.rgiovann.modelmapper.usecase.entities.Department;
import com.rgiovann.modelmapper.usecase.repositories.Departme

@Service
public class DepartmentService {

    @Autowired
    private DepartmentRepository repository;

    @Autowired
    private ModelMapper departmentModelMapper;

    public List<DepartmentDTO> findAll() {
        List<Department> list = repository.findAll(Sort.by("n
        return list.stream().map(department -> departmentMode
    }

    @Transactional
    public DepartmentDTO insert(DepartmentDTO departmentDTO)
        Department department = new Department();
        departmentModelMapper.map(departmentDTO,department );
        department = repository.save(department);
        departmentModelMapper.map(department, departmentDTO);
        return departmentDTO;
    }
}
```

A lógica em (A) é a mesma da lista paginada de empregados, com exceção que agora o retorno é uma C*ollection* do tipo *List*. Em (B) e (C) eu novamente populo a entidade com os atributos do DTO, aqui não há como popular os atributos da entidade que não estão mapeados no DTO (no caso *division* e *project)* que assumirão o valor *null* no banco de dados.

Uma observação, para utilizar o modelmapper, é necessário criar um *bean* que nada mais é do que indicar para o Spring “Hey *IOC container*, por gentileza faça a gestão desse objeto para mim, gerando todo o *boilerplate”* necessário para

instanciá-lo, para eu simplesmente usar quando necessário.”
Nessa caso é necessário adicionar a seguinte classe:

```
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ModelMapperConfig {

    @Bean
    ModelMapper modelMapper() {
        return new ModelMapper();
    }

}
```

Note que na camada de serviço eu não fiz nenhuma chamada as funções destacadas em vermelho na classe DTO, podem ser inclusive removidas, o *modelmapper* torna transparente a manipulação dos seus atributos. Esse artigo é introdutório e não pretende extinguir o uso da biblioteca, existem implementações mais avançadas quando por exemplo quisermos mapear atributos mais complexos (como uma Lista por exemplo).

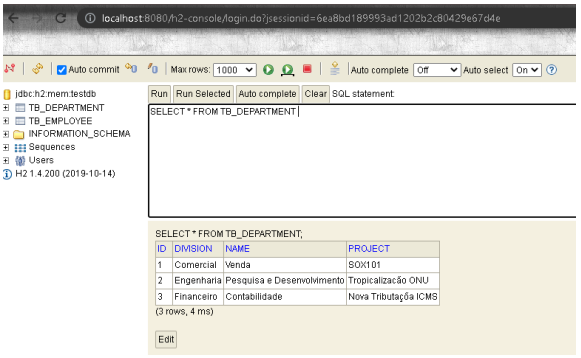
7.Populando o banco de dados H2

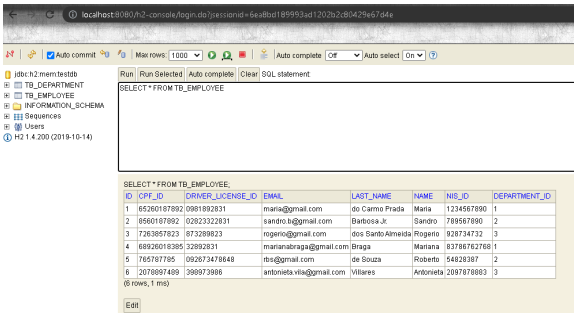
Em nosso exemplo usaremos o banco de dados H2, que é um banco de dados virtual gerado pelo próprio Spring Boot, muito útil enquanto ainda estamos na fase de desenvolvimento. No arquivo de *properties* do Spring Boot estão os comandos necessários para criar esse banco de dados, adicionalmente é necessário um arquivo de *seed* para popular o mesmo, no link gitub no final desse artigo está o código fonte gerado para ilustrar esse caso de uso.

O arquivo de *seed* utilizado está abaixo descrito:

```
INSERT INTO tb_department(name,division,project) VALUES ('Ven
INSERT INTO tb_department(name,division,project) VALUES ('Pes
INSERT INTO tb_department(name,division,project) VALUES ('Con

INSERT INTO tb_employee(name,last_name, email,driver_license_
INSERT INTO tb_employee(name,last_name, email,driver_license_
INSERT INTO tb_employee(name,last_name, email,driver_license_
INSERT INTO tb_employee(name,last_name, email,driver_license_
INSERT INTO tb_employee(name,last_name, email,driver_license_
INSERT INTO tb_employee(name,last_name, email,driver_license_
```

Que listando no banco H2:

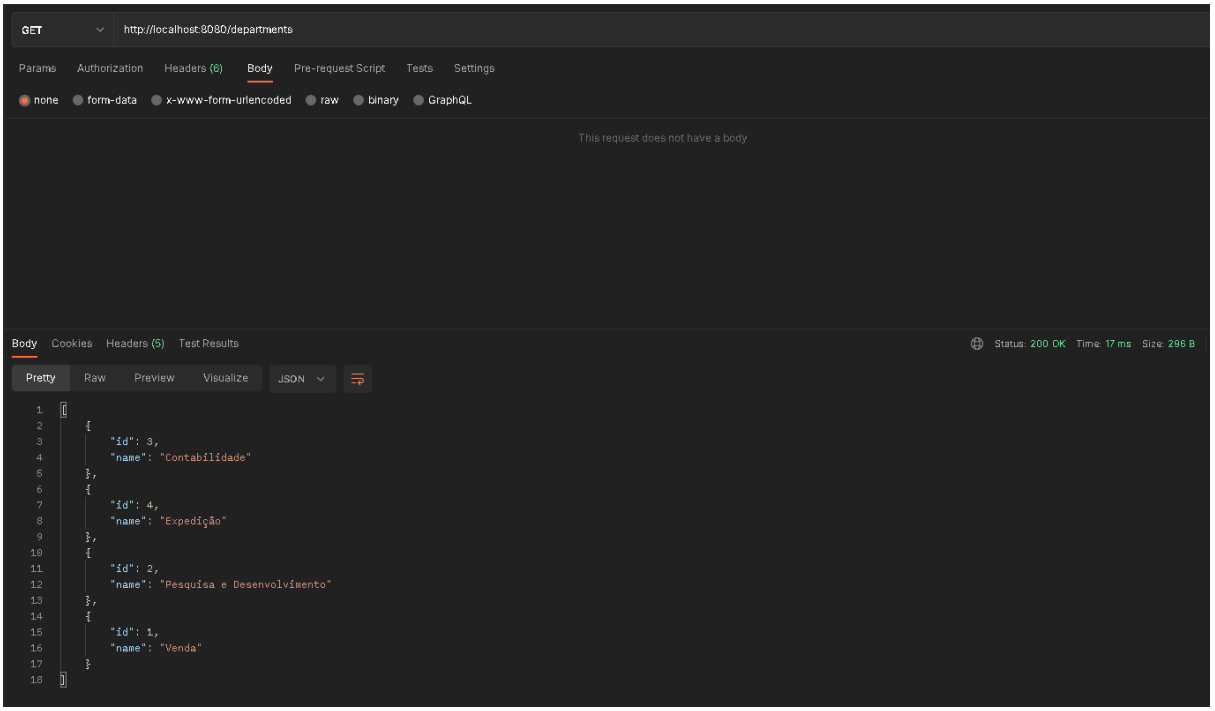




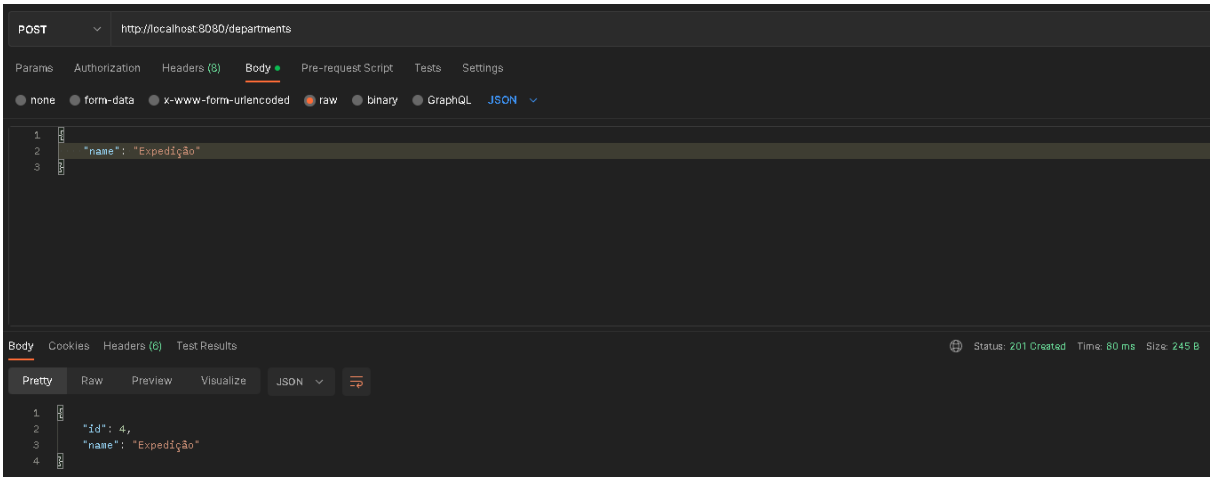
8.Gerando os *requests* para os *endopoints* via Postman

Os resultados do Postman para os *endpoints* definidos anteriormente:

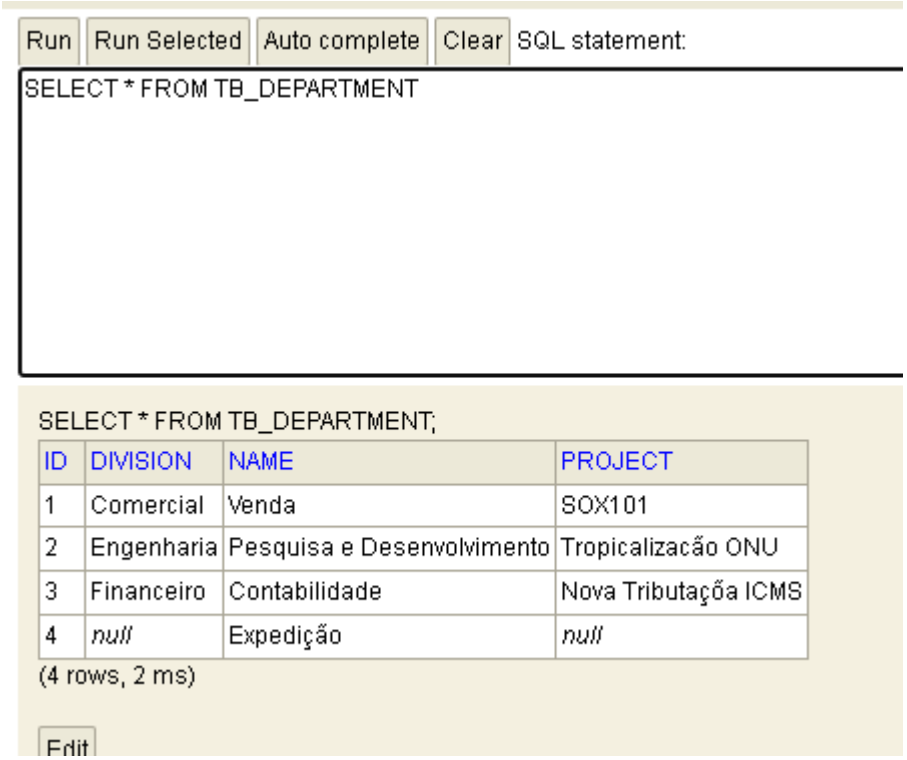
- **LISTA SIMPLES DE DEPARTAMENTOS**



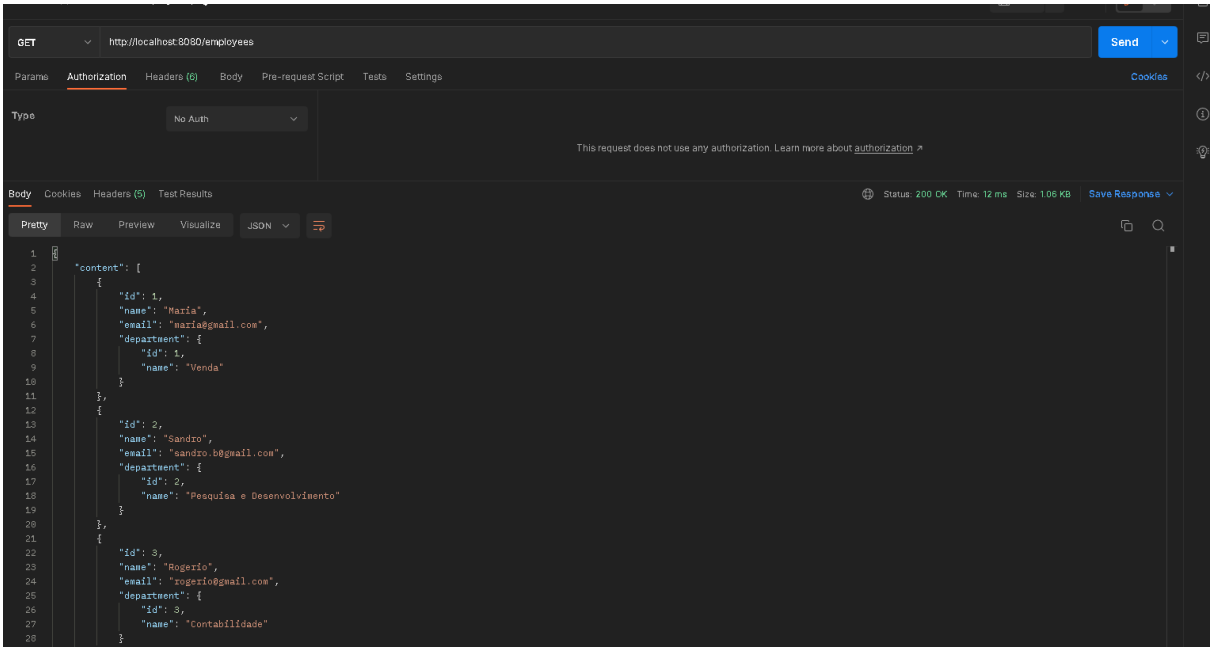
- **INSERÇÃO DE UM NOVO DEPARTAMENTO**



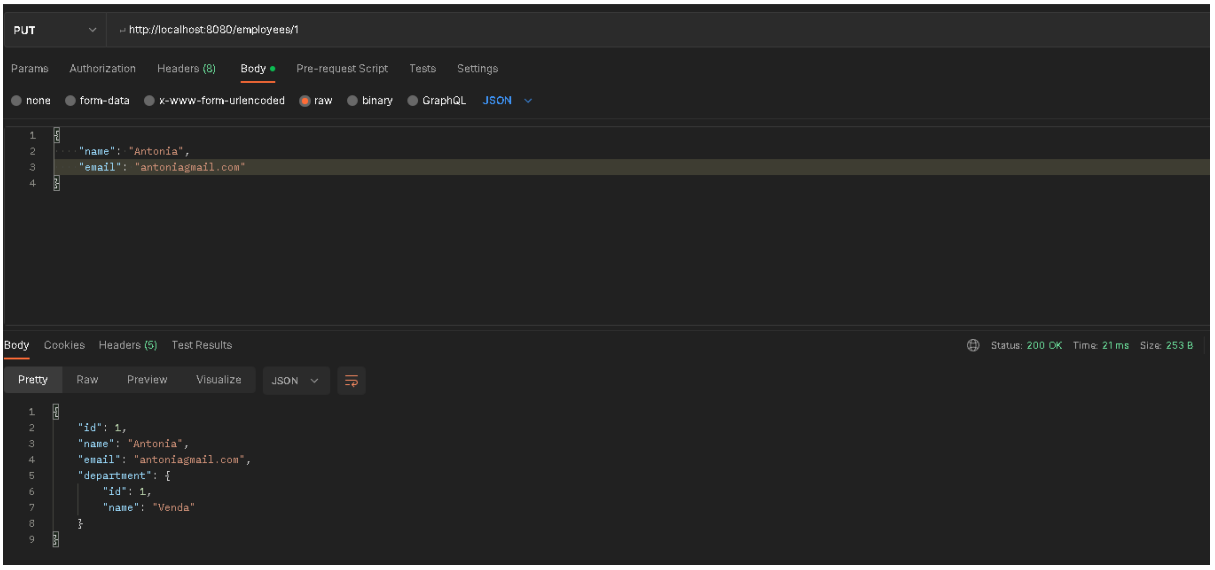
Olhando o banco de dados percebse-se que os outros campos foram criados como null, pois o DTO criado para este *endpoint* não continha essa informação:



- **LISTA PAGINADA DE TODOS OS EMPREGADOS**



- **ATUALIZAÇÃO DO NOME/EMAIL DO EMPREGADO**



Veja que apenas o nome/email foram atualizados, mesmo eu não fornecendo no body da mensagem HTTP (Json) o departamento do empregado

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM TB_EMPLOYEE
```

```
SELECT * FROM TB_EMPLOYEE;
```

ID	CPF_ID	DRIVER_LICENSE_ID	EMAIL	LAST_NAME	NAME	NIS_ID	DEPARTMENT_ID
1	65260187892	0981892831	antoniagmail.com	do Carmo Prada	Antonia	1234567890	1
2	8560187892	02823322831	sandro.b@gmail.com	Barbosa Jr.	Sandro	789567890	2
3	7263857823	873289823	rogerio@gmail.com	dos Santo Almeida	Rogerio	928734732	3
4	68926018385	32892831	marianabraga@gmail.com	Braga	Mariana	83786762768	1
5	765787785	092673478648	rbs@gmail.com	de Souza	Roberto	54828387	2
6	2078897489	398973986	antonieta.vila@gmail.com	Villares	Antonieta	2097878883	3

(6 rows, 2 ms)

Edit

9.Uma forma alternativa de mapeamento

Na lista paginada de empregados, a resposta contem o id do departamento uma vez que mapeamos o objeto *Department* no DTO da entidade *Employee.* Vamos supor que essa informação não é necessária, e apenas o nome do departamento é suficiente. Poderíamos substituir o atributo `department` que é um objeto do tipo `Department` por uma simples String na classe `EmployeeDTO`:

```
private String departmentName;
.
.
.
// não esquecendo de adicionar seus respectivos getter e sett

public EmployeeDTO(Long id, String name, String email, String
        this.id = id;
        this.name = name;
        this.email = email;
        this.departmentName = departmentName;
    }

public String getDepartmentName() {
        return departmentName;
    }

    public void setDepartmentName(String departmentName) {
        this.departmentName = departmentName;
    }
```

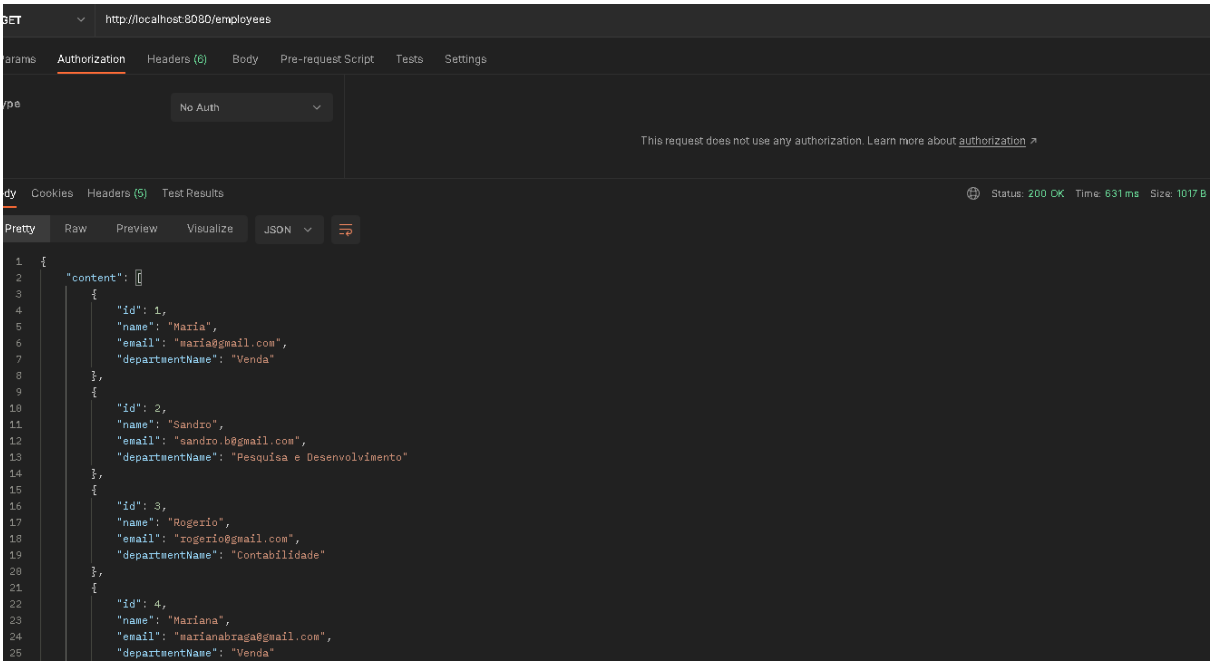
Neste caso precisamos informar ao *modelmapper* que um novo mapeamento (objeto *Department* –> String) deve ser realizado, no arquivo ***EmployeeService.java*** precisamos redefinir o mapeamento deste par através do *TypeMap* e uma função *lambda*:

```
@Transactional(readOnly = true)
    public  Page<EmployeeDTO> findAllPaged(Pageable pageReque
```

```
        // TypeMap encapusla as configurações de mapeamento do p
        // em nosso caso Employee.class (fonte) e EmployeeDTO.cl
               TypeMap<Employee, EmployeeDTO> propertyMapper = e

         // adiciona um novo mapeamento, definindo para o setter
         // obtido através de Employee.getDepartment().getName()

         propertyMapper.addMappings(
                         mapper -> mapper.map(src -> src.getDepa
                        );

               Page<Employee> page = employeeRepository.findAll(
               return page.map(p -> {EmployeeDTO employeeDTO = e
                                                  return employee
                                  }
                             );

      }
```

Dessa forma ao requisitarmos novamente a lista de empregados, trafegamos na rede apenas o

nome do departamento, economizando banda e tempo de processamento:



O código fonte utilizado para gerar esse artigo encontra-se no repositório abaixo:

```
https://github.com/rgiovann/modelmapper-use-cases
```