Spring Boot – Parte III

14 – Api de Notas

Na API de notas iremos novamente realizar as operações necessárias para realizar as operações de CRUD deste cadastro, teremos implementações dos verbos POST, PUT. DELETE e GET. Mas aqui teremos uma diferença importante, os relacionamentos de outras tabelas. Pois ao salvarmos dados de notas, será necessário as informações dos Alunos e Disciplinas anteriormente cadastrados.

Nosso trabalho começa, pela classe que vai representar o modelo de nosso cadastro de Notas. A ideia base será de informarmos o Aluno e a Disciplina e depois completarmos o cadastro com a informação da data e nota.

No pacote model.entity deve ser criado a classe Nota. Nesta classe teremos o @Entity, indicando que está classe será utilizada pelo JPA/Hibernate e as anotações do Lombok(@Getter e @Setter) para criar os métodos em tempo de execução.

Serão necessárias as informações de id, aluno, disciplina, data e valor da nota. O id será Integer com as anotações que o mesmo é o id da entidade e a sua estratégia de geração deste sequencial.

Para aluno será adicionado um atributo diretamente da classe Aluno, este atributo tem a anotação @ManyToOne, indicando que vários alunos podem ter notas e o @JoinColumn serve para indicar o relacionamento com a tabela de alunos e colocarmos o nome da coluna como id_aluno no banco de dados. A anotação @NotNull está indicando que o aluno é uma informação obrigatória.

Em disciplina será adicionado o atributo diretamente da classe disciplina, este atributo tem a anotação @ManyToOne, indicando que várias disciplinas podem ter notas e o @JoinColumn serve para indicar o relacionamento com a tabela de disciplinas e colocarmos o nome da coluna como id_disciplina. A anotação @NoNull está indicando que a disciplina é obrigatória.

Na data da nota, a anotação @Column define o nome da coluna no Banco de Dados e o @JsonFormat a formatação da data a ser retornada no JSON. O atributo nota, têm apenas uma validação indicando que a menor nota possível será 1 (um).

```java
@Entity
@Getter@Setter
public class Nota {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @ManyToOne
    @NotNull(message = "Deve ser informado o Aluno para indicar a nota!")
    @JoinColumn(name = "id_aluno")
    private Aluno aluno;

    @ManyToOne
    @NotNull(message = "Deve ser informada a Disciplina para indicar a nota!")
    @JoinColumn(name = "id_disciplina")
    private Disciplina disciplina;

    @Column(name = "data_nota")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataNota;

    @Column
    @Min(value = 1, message = "Menor nota é permitida é 1!")
    private BigDecimal nota;
}
```

Com o modelo pronto, deve ser criado o repository de Nota, que terá as operações de CRUD. Será criado em model.repository a Interface NotaRepository que vai herdar a JpaRepository e vai passar a informação da classe de modelo (Nota) e o tipo de dado do Id de Nota, neste caso será Integer.

```java
import com.bethaCode.alunos.model.entity.Nota;
import org.springframework.data.jpa.repository.JpaRepository;

public interface NotaRepository extends JpaRepository<Nota, Integer> {
}
```

Neste serviço teremos uma grande diferença, pois quando recebermos uma requisição do Browser, nosso serviço deve estar preparado para receber apenas o id do aluno, id da disciplina, além das informações da data e nota. Desta forma não podemos utilizar diretamente a entidade Nota para

receber o JSON, precisamos de uma classe intermediária (modelo de JSON), também conhecido como DTO.

Objeto de Transferência de Dados (do inglês, Data transfer object, ou simplesmente DTO), é um padrão de projeto de software, usado para transferir dados entre subsistemas de um software. DTOs são frequentemente usados em conjunção com objetos de acesso a dados, para obter informações de um banco de dados.

Então, deverá ser criado em model.dto a classe NotaDTO, ela será apenas criada para representar o JSON a ser recebido por nosso serviço. Esta classe terá a nota e data como String, pois iremos tratar estas informações em nosso Controller e as informações do idAluno e idDisciplina como Integer, pois receberemos apenas os códigos.

O construtor vazio, é uma necessidade do Spring Boot ao instanciar a classe.

```
@Getter@Setter
public class NotaDTO {
    private String nota;
    private String dataNota;
    private Integer idAluno;
    private Integer idDisciplina;

    public NotaDTO(){

    }
}
```

Lembrando que são transitados informações de texto entre o nosso backend e frontend, teremos o seguinte problema com dados numéricos.

O backend vai receber valores no formato 1.000,25, onde temos uma vírgula indicando o ponto flutuante e o . indicando a parte inteira. Porém em nosso banco de dados, o ponto flutuante é indicando por '.' e não são utilizados '.' para classificação dos valores inteiros.

Deve ser criado a classe BigDecimalConverter no pacote util. Esta classe tem a anotação @Component para indicarmos que a classe será adicionada pela injeção de dependência ao iniciar o

projeto. No método converter será recebida a String com o valor e apenas será alterada a formatação de acordo com a necessidade de nosso banco de dados

```
@Component
public class BigDecimalConverter {
    public BigDecimal converter(String valor){
        valor = valor.replace( target: ".",  replacement: "").replace( target: ",",  replacement: ".");
        return new BigDecimal(valor);
    }
}
```

No pacote rest, será criada a classe NotaController. Precisamos indicar para o Spring Boot que está classe vai receber requisições do nosso navegador e qual url será informada para executá-la.

Desta forma, deverá ser adicionada a anotação @RestController, indicando ao Spring Boot que esta classe receberá requisições rest e com a anotação @RequestMapping estaremos indicando o nome da url, neste caso caso será /api/notas. Como nossa aplicação está sendo executada na porta 8080, teremos http://localhost:8080/api/notas.

A anotação @RequiredArgsConstructor é do Lombok, serve para criar em tempo de execução um construtor com as variáveis necessárias para execução da classe. Neste caso vai utilizar os atributos que estão como final, assim gerando o construtor com os repositórios NotaRepository, AlunoRepository e DisciplinaRepository. Também com a classe utilitária BigDecimalConverter.

Para finalizar precisamos indicar o método e o verbo. Neste caso queremos criar uma nova nota em nosso banco, então devemos executar o método POST, desta forma adicionamos a anotação @PostMapping, que indica ao controller que este método será executado por uma requisição do tipo POST. Ainda temos a anotação @ResponseStatus, que vai indicar o tipo de retorno que será passado a requisição, quando executado com sucesso.

No método salvar, retornamos uma Nota e recebemos um objeto NotaDTO, porém no parâmetro temos a anotação @RequestBody. O @RequestBody está definindo que a nota será recebida no JSON no corpo da requisição Post.

Na execução do método temos uma tratativa diferente, pois recebemos um DTO, que precisamos converter no modelo de entidade, para realizar a persistência no Banco de Dados. Então

criamos uma nota e populamos com as informações de notaDTO, para depois realizar a persistência com a Nota.

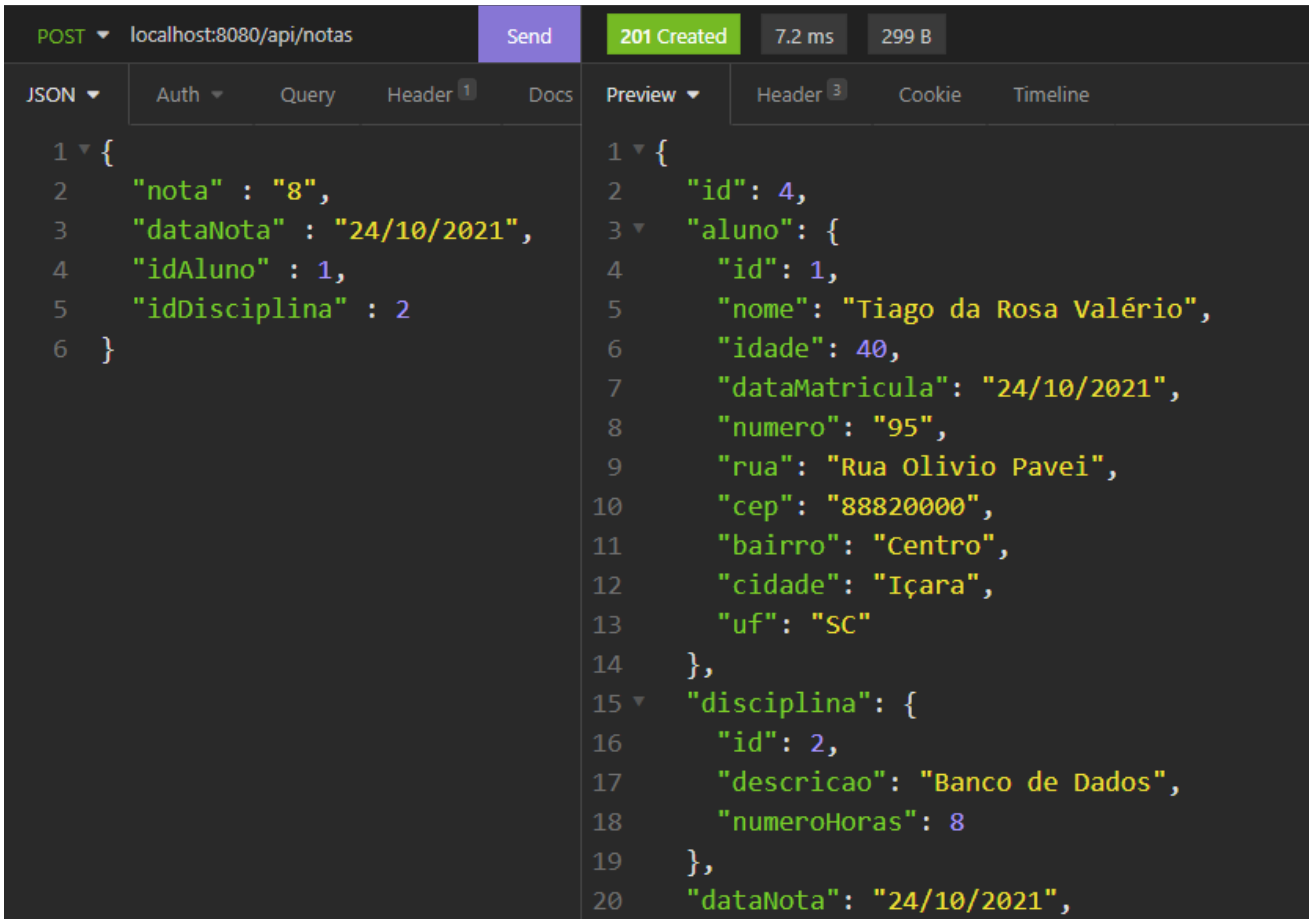
Na informação da data, é utilizado o LocalDate.parse, pegando a informação do dto da dataNota e indicando o formato que está data será recebida em nosso serviço. No valor da nota, utilizamos a nossa classe utilitária BigDecimalConverter para preparar a informação para o nosso modelo de nota.

Quanto ao aluno e disciplina recebemos o id, tentamos localizar a informação em seus repositórios, existindo adicionamos a informação ao modelo de Nota, senão existir geramos a exceção, indicando que a informação não existe em nossa aplicação.

```
@RestController
@RequestMapping("/api/notas")
@RequiredArgsConstructor
public class NotaController {

    private final NotaRepository notaRepository;
    private final AlunoRepository alunoRepository;
    private final DisciplinaRepository disciplinaRepository;
    private final BigDecimalConverter bigDecimalConverter;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Nota salvar(@RequestBody NotaDTO notaDTO){
        LocalDate dataNota = LocalDate.parse(notaDTO.getDataNota(), DateTimeFormatter.ofPattern("dd/MM/yyyy"));

        Integer idAluno = notaDTO.getIdAluno();
        Aluno aluno = alunoRepository
                        .findById(idAluno)
                        .orElseThrow(()-> new ResponseStatusException(HttpStatus.BAD_REQUEST,
                                "O aluno " + idAluno + " não existe em nossa aplicação!"));

        Integer idDisciplina = notaDTO.getIdDisciplina();
        Disciplina disciplina = disciplinaRepository
                                    .findById(idDisciplina)
                                    .orElseThrow(()-> new ResponseStatusException(HttpStatus.BAD_REQUEST,
                                            "A disciplina " + idDisciplina + " não existe em nossa aplicação!"));


        Nota nota = new Nota();
        nota.setDataNota(dataNota);
        nota.setAluno(aluno);
        nota.setDisciplina(disciplina);
        nota.setNota(bigDecimalConverter.converter(notaDTO.getNota()));

        return notaRepository.save(nota);
    }
}
```

No Insomnia precisamos criar uma requisição POST para notas, para isso precisamos apenas indicar a URL da requisição e montarmos o nosso JSON para envio na requisição. No caso o JSON são as informações dos atributos da classe NotaDTO.

Será enviado o JSON e receberemos como retorno, uma entidade de nota, porém com a informação do id, que foi gerado no banco de dados e o retorno do status é o 201 created, conforme configurado em nosso método executado pelo POST.



A próxima etapa de nosso servidor rest, será permitir a busca por notas cadastradas em nosso banco de dados, para isso utilizamos o verbo GET, será informado a URL base e o código da nota que desejamos a informação.

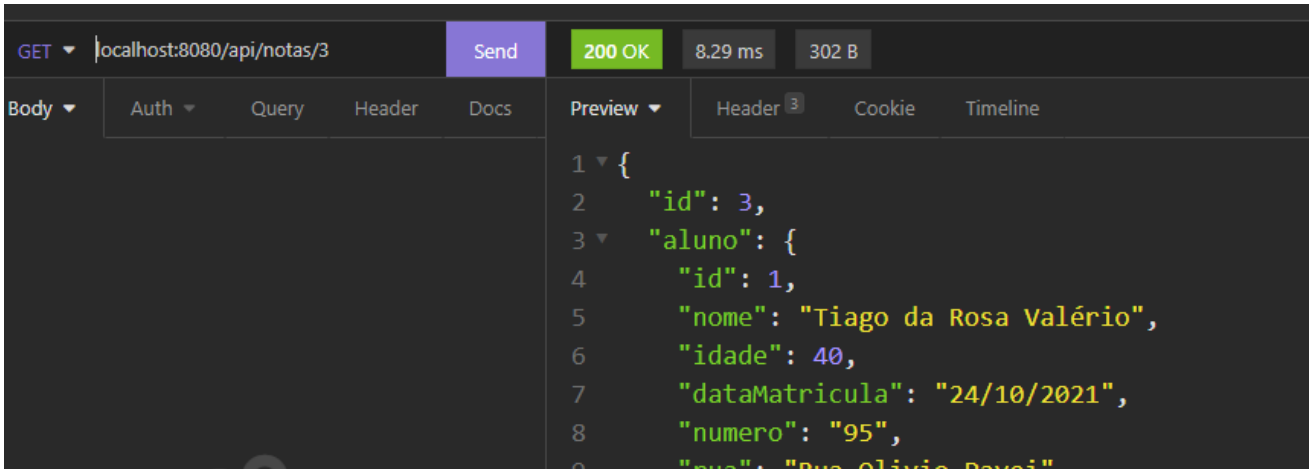
Então temos o método acharPorId que vai retornar uma Nota. A anotação @GetMapping está indicando que o método será executado, quando recebido o verbo GET nesta URL. Sendo indicando Id como parâmetro a ser recebido na requisição.

Como parâmetro temos a anotação @PathVariable indicando que será recebido um argumento pela requisição que será atribuído a variável id.

Para finalizar, nosso método utiliza o repositório executando a função findById passando o código da Nota, este metódo retorna um objeto do tipo Optional que pode ser uma Nota ou vazio. Será retornado a Nota caso exista, senão será executada a função orElseThrow onde estamos lançando uma exceção NOT_FOUND, indicando que a Nota não existe.

```
@GetMapping("{id}")
public Nota acharPorId(@PathVariable Integer id){
    return notaRepository
            .findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Nota " + id + " não cadastrada!"));
}
```

Para testar o método GET, precisamos criar uma nova requisição no Insomnia com o verbo GET e na url de notas adicionar / e o código da nota que desejamos selecionar. Neste caso será retornado a Nota ou Not Found quando o mesmo não existir em nosso banco de dados.



O processo de exclusão será basicamente o mesmo que a busca da Nota, porém iremos utilizar o método para exclusão do repositório.

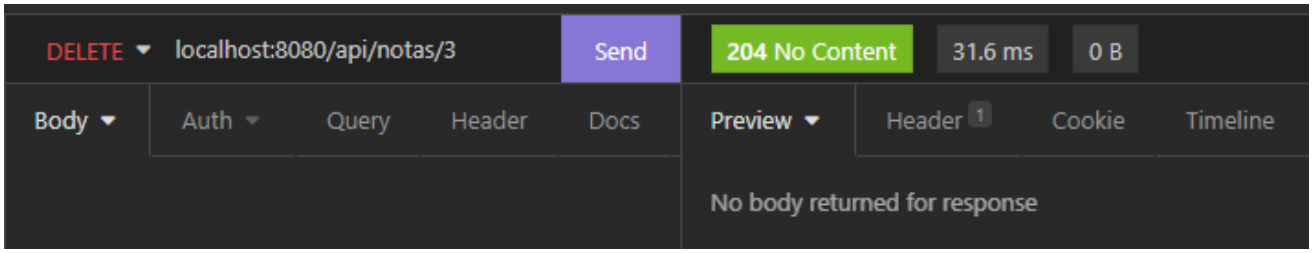
Temos o método deletar sem retorno, que vai receber o parâmetro passado na URL com o nome de id. Utilizamos a anotação @DeleteMapping para indicar que este método deve ser executado ao receber uma requisição do método DELETE.

A anotação @ResponseStatus vai retornar NO_CONTENT, quando a exclusão for executada com sucesso, pois este é o retorno a ser utilizado de forma padrão para este tipo de operação. No

corpo do método utilizamos o findById quando localizado temos o retorno em map, onde pegamos a nota retornada, excluimos e adicionamos o retorno void. Quando não localizado a nota, retornamos o status NOT_FOUND para a requisição.

```
@DeleteMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deletar(@PathVariable Integer id){
    notaRepository
            .findById(id)
            .map(nota -> {
                notaRepository.delete(nota);
                return Void.TYPE;
            })
            .orElseThrow(()-> new ResponseStatusException(HttpStatus.NOT_FOUND, "Nota " + id + " não cadastrada!"));
}
```

Para testar o método DELETE, precisamos criar uma nova requisição no Insomnia com o verbo DELETE e na url de notas adicionar / e o código da nota que desejamos selecionar. Neste caso será retornado No Content quando excluído ou Not Found quando o mesmo não existir em nosso banco de dados.



Nosso último método será o atualizar, este será parecido com o Adicionar, pois precisaremos passar o JSON com as alterações realizadas na Nota para o método.

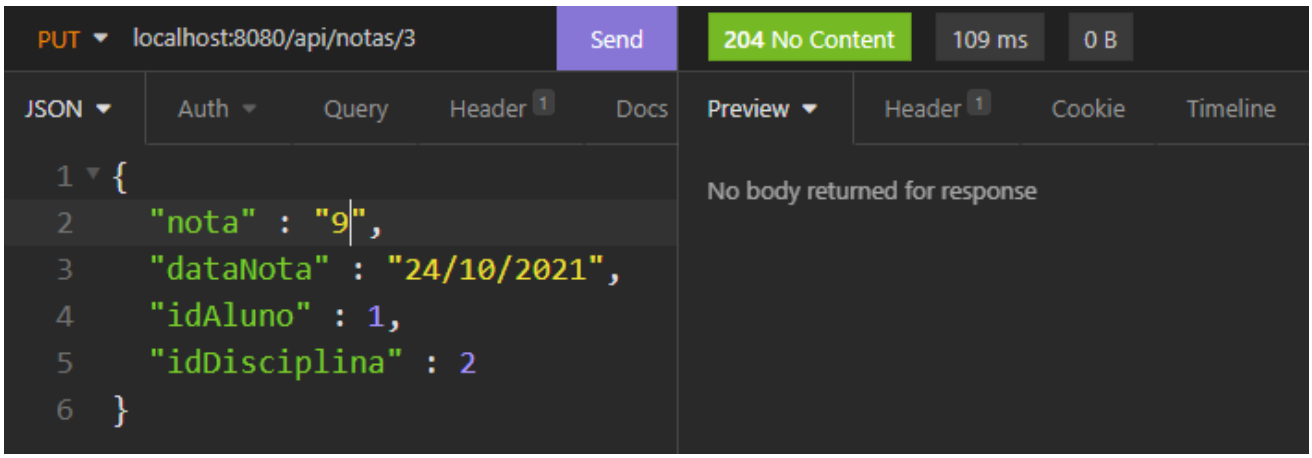
A anotação @PutMapping indica que o método será executado quando realizada uma requisição com o verbo PUT, sendo que teremos dois parâmetros. A informação do Id que queremos atualizar e um JSON com as informações da Nota, por isso nosso parâmetro possui a anotação @RequestBody, indicando que a informação da nota será recebida no corpo da requisição.

No corpo do método utilizamos o findById, quando localizado a nota, setamos nele as informações recebidas na requisição e salvamos o cadastro, quando não existir será retornado o Status NOT_FOUND.

```java
@PutMapping("{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void atualizar(@PathVariable Integer id, @RequestBody NotaDTO dadoAtualizado){

    LocalDate dataNota = LocalDate.parse(dadoAtualizado.getDataNota(), DateTimeFormatter.ofPattern("dd/MM/yyyy"));

    Integer idAluno = dadoAtualizado.getIdAluno();
    Aluno aluno = alunoRepository
            .findById(idAluno)
            .orElseThrow(()-> new ResponseStatusException(HttpStatus.BAD_REQUEST,
                    "O aluno " + idAluno + " não existe em nossa aplicação!"));

    Integer idDisciplina = dadoAtualizado.getIdDisciplina();
    Disciplina disciplina = disciplinaRepository
            .findById(idDisciplina)
            .orElseThrow(()-> new ResponseStatusException(HttpStatus.BAD_REQUEST,
                    "A disciplina " + idDisciplina + " não existe em nossa aplicação!"));

    notaRepository
            .findById(id)
            .map(nota -> {
                nota.setDataNota(dataNota);
                nota.setNota(bigDecimalConverter.converter(dadoAtualizado.getNota()));
                nota.setDisciplina(disciplina);
                nota.setAluno(aluno);
                return notaRepository.save(nota);
            })
            .orElseThrow(()-> new ResponseStatusException(HttpStatus.NOT_FOUND, "Nota " + id + " não cadastrada!"));
}
```

Para testar nosso PUT, deve ser criada uma nova requisição do tipo PUT, indicando que no corpo da requisição teremos um JSON.

Será necessário indicar na URL o id que queremos atualizar e informar o JSON com os dados que queremos alterar, não é necessário informar o ID, pois é o código informado na requisição.



Ainda iremos criar um método GET, que permita passarmos o nome do Aluno e este vai retornar uma lista com as notas deste aluno.

Para isso, precisamos criar um novo método em nosso repository de Notas, que permita uma consulta com o parâmetro do nome do aluno.

Em NotaRepository iremos adicionar o método findByNomeAluno que vai retornar uma lista de alunos e vai receber o parâmetro nome, para isso temos a anotação do @Param. A anotação @Query está indicando para o Spring Boot que se refere a um JPQL. Este vai buscar as notas relacionadas ao aluno que possui o nome como o passado no parâmetro.

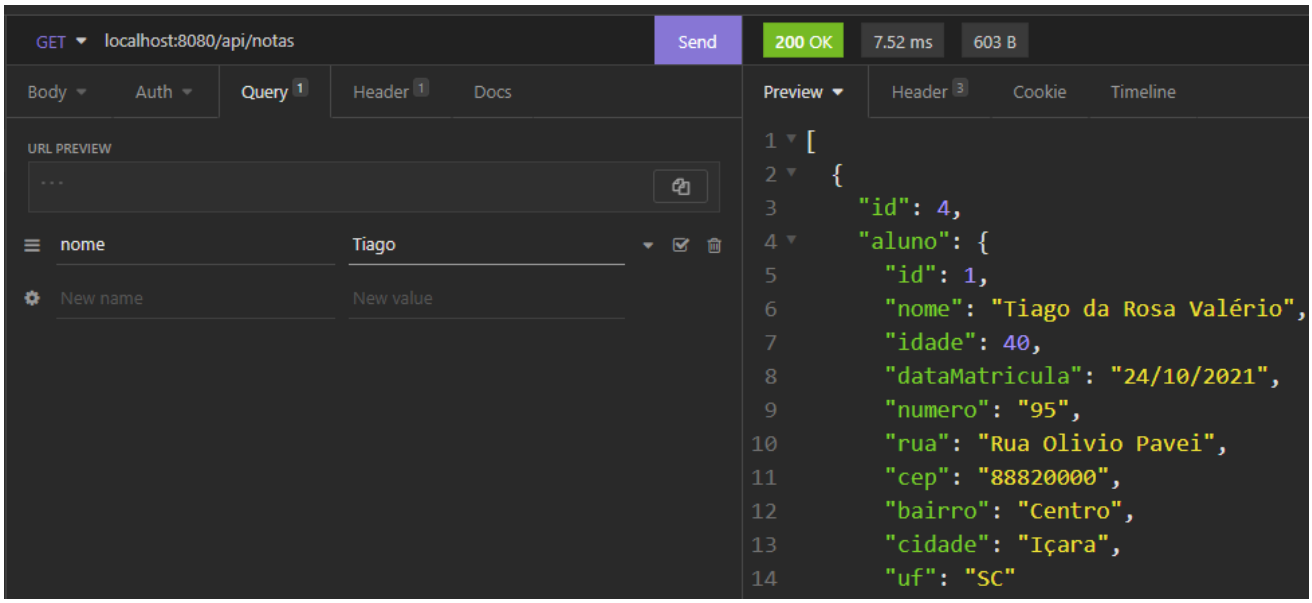
```
public interface NotaRepository extends JpaRepository<Nota, Integer> {

    @Query(" select n from Nota n join n.aluno a where upper(a.nome) like upper(:nome) ")
    List<Nota> finByNomeAluno(@Param("nome") String nome);

}
```

Na Controller de Notas, iremos adicionar um novo método para o verbo GET, porém este não têm anotação que receberá o ID, desta forma será executado diretamente em api/notas, sem a informação do código.

Porém adicionamos em seu parâmetro a anotação @RequestParam, indicando que na requisição será passada a informação do parâmetro nome. Com este nome recebido, retornamos uma lista utilizando a nossa consulta JPQL adicionada na repository.

```
@GetMapping
public List<Nota> pesquisar(
        @RequestParam(value = "nome", required = false, defaultValue = "") String nomeDoAluno){
    return notaRepository.finByNomeAluno("%" + nomeDoAluno + "%");
}
```

No Insomnia adicionaremos uma nova requisição GET para Notas, porém na URL não teremos a informação do ID. Mas na guia Query iremos adicionar o parâmetro nome e a informação que queremos filtrar em nosso serviço.


GET
localhost:8080/api/notas
Send
200 OK
7.52 ms
603 B
Body
Auth
Query 1
Header 1
Docs
Preview
Header 3
Cookie
Timeline
URL PREVIEW
nome
Tiago
New name
New value
[
{
"id": 4,
"aluno": {
"id": 1,
"nome": "Tiago da Rosa Valério",
"idade": 40,
"dataMatricula": "24/10/2021",
"numero": "95",
"rua": "Rua Olivio Pavei",
"cep": "88820000",
"bairro": "Centro",
"cidade": "Içara",
"uf": "SC"