INSTITUTO FEDERAL
Triângulo Mineiro
Campus Uberlândia Centro

Projeto Backend Microsserviços e NoSQL

Integrando projeto ao MongoDB

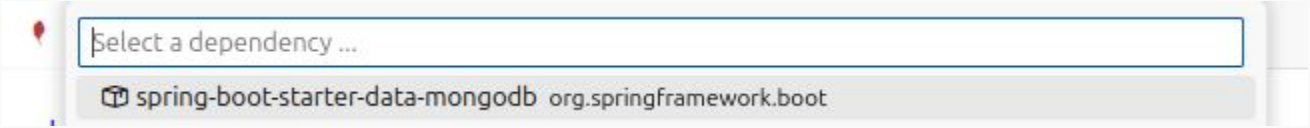
Prof. Lucas Montanheiro
lucasmontanheiro@iftm.edu.br

Comandos para subir o MongoDB

- Vamos utilizar o MongoDB Atlas.
- Pegue a usa URI e senha de conexão ao MongoDB para seguirmos.

Instalando Dependência do MongoDB

- Para começar vamos adicionar ao nosso projeto a dependência do Spring Boot chamada: **spring-boot-starter-data-mongodb**



Configurando o application.properties

- Adicione as informações do banco de dados MongoDB no:
- resources > application.yml
- Caso não tenha o arquivo .yml, crie e apague o .properties

src > main > resources > application.yml

```yaml
spring:
  data:
    mongodb:
      uri: mongodb+srv://lucasmontanheiro:****@cluster0.7m1i0.mongodb.net/
      database: user-api
```

Primeiros passos com MongoDB no Java

- Em nossa classe principal do projeto, precisamos adicionar a notação **@EnableMongoRepositories**, que automaticamente será feita a importação do Spring Boot Data com MongoDB.

src > main > java > com > montanha > user_api > UserApiApplication.java > Language Support for Java(TM) by Red Hat > UserApiApplication

```java
package com.montanha.user_api;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;

@SpringBootApplication
@EnableMongoRepositories
public class UserApiApplication {

    Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(UserApiApplication.class, args);
    }

}
```

Camada de Dados (Repository)

- Vamos criar uma nova pasta no main do projeto chamada **models** e mover a pasta **dto** para dentro dela.
- Agora sempre que tivermos uma classe DTOs, também será necessário criar uma entidade, que é o objeto que possui exatamente a mesma estrutura do banco de dados.
- Essas classes são anotadas com **@Document**, indicando que elas representam um documento dentro de uma coleção no MongoDB.

Camada de Dados (Repository)

- Nessa classe, o ID da entidade é marcado com a anotação **@Id**, que indica que o campo é o identificador único do documento.
- No MongoDB, o campo **_id** é gerado automaticamente e, por padrão, é do tipo **ObjectId**.
- Se necessário, você pode configurar manualmente o tipo do campo ou gerar IDs personalizados.

Camada de Dados (Repository)

- Além disso, os outros atributos podem ser marcados com a anotação **@Field**, que possui propriedades como se o campo é obrigatório ou não, o nome do campo no documento, e outras configurações.
- Assim como na classe UserDTO, iremos adicionar as anotações do **Lombok** na classe User para simplificar a criação de **getters**, **setters** e **construtores**.
- Vamos criar a **User.class** dentro de **models**:

Classe User em models

src > main > java > com > montanha > user_api > models > User.java > ...

```java
package com.montanha.user_api.models;

import java.time.LocalDateTime;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;
import com.montanha.user_api.models.dto.UserDTO;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "user")
public class User {

    @Id
    private String id;
    @Field("nome")
    private String nome;
    @Field("cpf")
    private String cpf;
    @Field("endereco")
    private String endereco;
    @Field("email")
    private String email;
    @Field("telefone")
    private String telefone;
    @Field("dataCadastro")
    private LocalDateTime dataCadastro;

    public static User convert(UserDTO userDTO) {
        User user = new User();
        user.setNome(userDTO.getNome());
        user.setEndereco(userDTO.getEndereco());
        user.setCpf(userDTO.getCpf());
        user.setEmail(userDTO.getEmail());
        user.setTelefone(userDTO.getTelefone());
        user.setDataCadastro(userDTO.getDataCadastro());
        return user;
    }

}
```

Camada de Dados (Repository)

- O método convert, na classe **User**, é importante porque precisaremos converter instâncias da entidade **User** para instâncias da classe **UserDTO**.
- Isso será utilizado em diversos serviços, então é melhor ter um método que realiza essa operação do que ficar duplicando código em vários locais.
- A mesma coisa será necessária na classe **UserDTO** , como podemos ver a seguir:

Classe UserDTO em models > dto

src > main > java > com > montanha > user_api > models > dto > J UserDTO.java > ...

```java
package com.montanha.user_api.models.dto;

import java.time.LocalDateTime;

import com.montanha.user_api.models.User;
import jakarta.validation.constraints.NotBlank;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class UserDTO {

    private String id;
    @NotBlank (message = "Nome é obrigatório")
    private String nome;
    @NotBlank(message = "CPF é obrigatório")
    private String cpf;
    private String endereco;
    @NotBlank(message = "E-mail é obrigatório")
    private String email;
    private String telefone;
    private LocalDateTime dataCadastro;

    public static UserDTO convert(User user) {
        UserDTO userDTO = new UserDTO();
        userDTO.setId(user.getId());
        userDTO.setNome(user.getNome());
        userDTO.setEndereco(user.getEndereco());
        userDTO.setCpf(user.getCpf());
        userDTO.setEmail(user.getEmail());
        userDTO.setTelefone(user.getTelefone());
        userDTO.setDataCadastro(user.getDataCadastro());
        return userDTO;
    }
}
```

Camada de Dados (Repository)

- Com o Spring Data MongoDB, agora vamos criar um repositório para a entidade criada.
- Primeiro vamos criar a pasta **repositories** e dentro da pasta a interface **UserRepository.java**.
- O repositório é uma interface anotada com **@Repository,** que também é um **bean** do Spring e será automaticamente instanciado na inicialização da aplicação.

Camada de Dados (Repository)

- Essa interface deve estender **MongoRepository**, passando a entidade **User** e o tipo do **ID**, que, no seu caso, é **ObjectId**.
- O código da interface **UserRepository** pode será da seguinte forma:

src > main > java > com > montanha > user_api > repositories > J UserRepository.java > ...

```java
package com.montanha.user_api.repositories;

import org.bson.types.ObjectId;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;
import com.montanha.user_api.models.User;

@Repository
public interface UserRepository extends MongoRepository<User, String> {

}
```

Camada de Dados (Repository)

- Podemos adicionar também alguns métodos como **findByCpf** e **queryByNameLike**, algumas consultas podem ser criadas apenas com o nome do método.
- Esses métodos devem ter algumas palavras-chaves no nome como **find**, **and**, **or**, **like** e o nome do campo.
- Nos próximos serviços, criaremos algumas consultas mais complexas, e vamos ver mais dessas palavras-chaves.

Camada de Dados (Repository)

src > main > java > com > montanha > user_api > repositories > UserRepository.java > ...

```java
package com.montanha.user_api.repositories;

import java.util.List;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;
import com.montanha.user_api.models.User;

@Repository
public interface UserRepository extends MongoRepository<User, String> {

    User findByCpf(String cpf);

    List<User>  queryByNomeLike(String  name);

}
```

Camada de Serviços (Service)

- O **controller** poderia chamar o repositório diretamente, porém a maioria das aplicações possui uma camada intermediária, chamada de **service**, que é onde ficam as regras de negócio da aplicação.
- Essas classes devem ser anotadas com **@Service** e normalmente são responsáveis por fazer chamadas ao repositório e também a outros serviços.

Camada de Serviços (Service)

- Vamos criar em nosso projeto a pasta **services** e a classe **UserService.java**.
- A listagem mostra um exemplo de uma classe **Service** que será utilizada para acessar os dados da collection **user**.

src > main > java > com > montanha > user_api > services > UserService.java > ...

```java
package com.montanha.user_api.services;

import java.util.List;
import java.util.stream.Collectors;
import org.springframework.stereotype.Service;
import com.montanha.user_api.models.dto.UserDTO;
import com.montanha.user_api.models.User;
import com.montanha.user_api.repositories.UserRepository;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;

    public List<UserDTO> getAll() {
        List<User> usuarios = userRepository.findAll();
        return usuarios.stream()
                .map(UserDTO::convert)
                .collect(Collectors.toList());
    }
}
```

Camada de Serviços (Service)

- Na listagem, é possível ver que a classe foi anotada com **@Service**, indicando que uma instância dela será criada na criação da aplicação.
- A classe também possui um atributo do tipo **UserRepository**.
- Sem o Lombok é necessário usar o anotação **@Autowired**, que serve para fazer injeção de dependências.
- Porém, usando o Lombok, poderemos remover todas essas anotações e substituí-las pela anotação **@RequiredArgsConstructor**, que fica na classe.
- Com ela, evitamos ter que utilizar essa anotação **@Autowired** em classes que possuem a injeção de diversas dependências.

Camada de Serviços (Service)

Além disso, a classe possui o método **getAll**, que faz as seguintes operações:

- Chama o método **findAll**, do **UserRepository**, que retorna uma lista de usuários, sendo instâncias da entidade **User**;
- Transforma a lista em um **stream** e chama o método **map** para transformar a lista de entidades em uma lista de DTOs;
- Retorna a lista de DTOs.

Camada de Serviços (Service)

- Além do método **getAll**, adicionaremos mais seis métodos nesta classe: **findById**, **save**, **delete**, **findByCpf**, **queryByName** e **editUser**.
  - O **findById** busca um usuário por um id específico;
  - O **save** salva um usuário no banco de dados;
  - O **delete** exclui um usuário do banco de dados;
  - O **findByCpf** faz a busca de um usuário por seu CPF;
  - O **queryByName** faz uma busca pelo nome do usuário, mas, diferentemente da busca por id ou pelo CPF, a busca não será exata, mas sim pela inicial do nome passada no parâmetro. Por exemplo, se o parâmetro nome tiver valor **Mar%** , a busca retornará pessoas com o nome Marcela, Marcelo ou Marcos.
  - O **editUser**, receberá o id de um usuário e um objeto da classe UserDTO. Com o id, será buscado um usuário no banco de dados e, caso ele exista, os dados desse usuário serão atualizados com o que foi recebido no objeto do tipo UserDTO; caso o usuário não exista, será retornado um erro de que o usuário não foi encontrado.

Implementar na UserService.java

```java
    public UserDTO findById(String userId) {
        User usuario = userRepository.findById(userId)
                .orElseThrow(() -> new RuntimeException("User not found"));
        return UserDTO.convert(usuario);
    }

    public UserDTO save(UserDTO userDTO) {
        userDTO.setDataCadastro(LocalDateTime.now());
        User user = userRepository.save(User.convert(userDTO));
        return UserDTO.convert(user);
    }
    public UserDTO delete(String userId) {
        User user = userRepository
            .findById(userId).orElseThrow(() -> new RuntimeException());
                    userRepository.delete(user);
                    return UserDTO.convert(user);
    }
    public UserDTO findByCpf(String cpf) {
        User user = userRepository.findByCpf(cpf);
        if (user != null) {
            return UserDTO.convert(user);
        }
        return null;
    }

    public List<UserDTO> queryByName(String name) {
    List<User> usuarios = userRepository.queryByNomeLike(name);
    return usuarios
        .stream()
        .map(UserDTO::convert)
        .collect(Collectors.toList());
    }

```

```java

    public UserDTO editUser(String userId, UserDTO userDTO) {
        User user = userRepository
                .findById(userId).orElseThrow(() -> new RuntimeException());
        if (userDTO.getEmail() != null &&
                !user.getEmail().equals(userDTO.getEmail())) {
            user.setEmail(userDTO.getEmail());
        }
        if (userDTO.getTelefone() != null &&
                !user.getTelefone().equals(userDTO.getTelefone())) {
            user.setTelefone(userDTO.getTelefone());
        }
        if (userDTO.getEndereco() != null &&
                !user.getEndereco().equals(userDTO.getEndereco())) {
            user.setEndereco(userDTO.getEndereco());
        }
        user = userRepository.save(user);
        return UserDTO.convert(user);
    }

    public Page<UserDTO> getAllPage(Pageable page) {
        Page<User> users = userRepository.findAll(page);
        return users
            .map(UserDTO::convert);
    }

}

```

Camada de Serviços (Service)

- O Spring possui uma funcionalidade bastante interessante, que é a paginação.
- É bastante simples de implementar utilizando o Spring Data, que já disponibiliza duas interfaces prontas para serem usadas nesses casos: o **Page** e o **Pageable**.

Camada de Serviços (Service)

- O **Page** é uma coleção de objetos já paginados, que possui dados como o tamanho de uma página, quantos existem no total e qual a página que o usuário está visualizando.
- Já o **Pageable** recebe os parâmetros que o usuário pode usar para acessar uma página, como qual o tamanho e qual a página que deve ser acessada.
- A interface **JpaRepository** já possui um método **findAll** que recebe um objeto **Pageable** e faz a consulta fazendo paginação.
- Vamos implementar a consulta com paginação:

Camada de Serviços (Service)

```
8   import org.springframework.data.domain.Page;
9   import org.springframework.data.domain.Pageable;
```

```
81
82      public Page<UserDTO> getAllPage(Pageable page) {
83          Page<User> users = userRepository.findAll(page);
84          return users
85              .map(UserDTO::convert);
86      }
87
88  }
89
```

Camada dos Controladores (Controllers)

- Os **controllers** mudam pouco em relação ao que já havíamos feito.
- Basicamente, eles chamarão a classe da camada de serviço.
- A classe continua com a mesma anotação **@RestController**, e os métodos com as anotações **@GetMapping**, **@PostMapping** e **@DeleteMapping**.
- Uma diferença é a injeção da dependência da classe de serviços **UserService**.

Camada dos Controladores (Controllers)

- Os métodos nela são os mesmos, com a diferença de que agora são chamados os métodos da camada de serviço em vez de manipular a lista em memória.
- Também vamos mudar o nome de algumas rotas para ficar mais próximo do que usaremos nos outros serviços; por exemplo:
    - **DELETE /removeUser/{cpf}** virou **DELETE /user/{id}** .

Camada dos Controladores (Controllers)

- Outro ponto importante nas rotas são os códigos de retorno.
- Por padrão, as rotas REST retornam o código 200, porém é uma boa prática mudar o tipo de retorno dependendo do tipo da rota.
- Nas rotas que criam um objeto, o melhor código de retorno é o 201, que indica que um recurso foi criado no servidor;
- Para uma rota DELETE, normalmente é retornado um código 204, indicando que a rota não terá nenhum dado de retorno.

Camada dos Controladores (Controllers)

```java
package com.montanha.user_api.controller;

import java.util.List;

import org.bson.types.ObjectId;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

import com.montanha.user_api.models.dto.UserDTO;
import com.montanha.user_api.services.UserService;

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;

@RestController
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {

```

```java
public class UserController {

    private final UserService userService;

    @GetMapping
    public List<UserDTO> getUsers() {
        return userService.getAll();
    }

    @GetMapping("/{id}")
    public UserDTO findById(@PathVariable  String id) {
        return userService.findById(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public UserDTO newUser(@RequestBody @Valid UserDTO userDTO) {

        return userService.save(userDTO);
    }
    @GetMapping("/{cpf}/cpf")
    public UserDTO findByCpf(@PathVariable String cpf) {
        return userService.findByCpf(cpf);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void delete(@PathVariable  String id) {
        userService.delete(id);
    }

    @GetMapping("/search")
      public List<UserDTO> queryByName(
          @RequestParam(name="nome", required = true) String nome) {
        return userService.queryByName(nome);
    }
}
```

Camada dos Controladores (Controllers)

- Uma novidade aqui é a rota **/user/search**, que fará a busca pelo nome recebido como parâmetro - o nome pode ser completo ou apenas parte do nome.
- Se o nome for completo, a rota retornará apenas um usuário;
- Se o usuário passar apenas parte do nome, pode ser retornada uma lista de usuários.

Camada dos Controladores (Controllers)

- Outra novidade é a anotação **@RequestParam**, que deve ser usada quando queremos passar parâmetros na URL para a rota.
- Veja que a anotação recebeu que o parâmetro é obrigatório.

Camada dos Controladores (Controllers)

- A chamada para essa rota pode ser feita pela URL http://localhost:8080/user/search?nome=mar
- Nesse caso, a resposta para a chamada será:


```
[
    {
        "nome": "marcela",
        "cpf": "123",
        "endereco": "Rua abc",
        "email": "marcela@email.com",
        "telefone":"1234-3454",
        "dataCadastro": "2019-11-17T21:04:51.701+0000",
    },
    {
        "nome": "marcelo",
        "cpf": "123",
        "endereco": "Rua abc",
        "email": "marcelo@email.com",
        "telefone":"1234-3454",
        "dataCadastro": "2019-11-17T21:04:51.701+0000",
    }
]
```

Camada dos Controladores (Controllers)

- Além da rota de **search**, também foram adicionadas outras duas rotas no **UserController**: as rotas de edição de usuário e a rota que retorna os usuários usando paginação.
- Agora vamos implementar a rota de edição de usuários.
- A rota de edição é parecida com a de criação de usuário.
- Ela recebe um JSON que representa um usuário e atualiza os dados do usuário conforme o que foi recebido.

Camada dos Controladores (Controllers)

```
54
55      @PatchMapping("/{id}")
56      public UserDTO editUser(@PathVariable  String  id,
57              @RequestBody UserDTO userDTO) {
58          return userService.editUser(id, userDTO);
59      }
60  }
```

Camada dos Controladores (Controllers)

- Com essa implementação podemos atualizar o endereço do usuário que tem o CPF **123**, podemos chamar a rota PATCH http://localhost:8080/user/**123**, passando apenas os dados que queremos atualizar, como o endereço e o telefone:

```
{
    "endereco": "Rua abc",
    "telefone":"1234-3454"
}
```

Camada dos Controladores (Controllers)

- Agora vamos implementar a busca de usuários usando paginação.
- A chamada é feita na rota GET http://localhost:8080/user/pageable?size=2&page=1.
- Note os parâmetros size e page.
- O size indica que as páginas terão tamanho 2, e que deve ser retornada a página 1 (lembrando que as páginas começam no índice 0).
- A listagem a seguir mostra o código dessa rota.

Camada dos Controladores (Controllers)

```
60
61      @GetMapping("/pageable")
62      public Page<UserDTO> getUsersPage(Pageable pageable) {
63          return userService.getAllPage(pageable);
64      }
65  }
66
```

Camada dos Controladores (Controllers)


```
{
    "content": [
        {
            "id": "67325e5521d198418b5a9d41",
            "nome": "João",
            "cpf": "123",
            "endereco": "Avenida Getulio Vargas",
            "email": "joao@maria.com",
            "telefone": "12345-9999",
            "dataCadastro": "2024-11-11T16:43:17.503"
        },
        {
            "id": "67325e6421d198418b5a9d42",
            "nome": "João",
            "cpf": "123",
            "endereco": "Avenida Rondom",
            "email": "joao@maria.com",
            "telefone": "99999-9999",
            "dataCadastro": "2024-11-11T16:43:32.572"
        }
    ],
    "pageable": {
        "pageNumber": 0,
        "pageSize": 2,
        "sort": {
            "sorted": false,
            "empty": true,
            "unsorted": true
        },
        "offset": 0,
        "paged": true,
        "unpaged": false
    },
    "totalPages": 4,
    "totalElements": 7,
    "last": false,
    "size": 2,
    "number": 0,
```

Camada dos Controladores (Controllers)

- No retorno, existe um objeto chamado **content**, que possui a lista de usuários;
- Depois, existem diversos dados sobre paginação, como o número de página, o número total de elementos e o tamanho da página.
- Esses dados podem ser utilizados no front-end para a criação de uma tabela com paginação.
- Agora já temos uma versão funcional do user-api com integração com o banco de dados MongoDB.