INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO, CAMPUS CUBATÃO

TUTORIAL G1 – REACT + EXPRESS (NODEJS) + MONGODB

Beatriz Bastos Borges

Beatriz Pereira Carlos

Laysa Bernardes Campos da Rocha

Lucas Lopes Cruz

Maria Eduarda Fodor

Miguel Luizatto Alves

Ricardo Queiroz Oliani

CUBATÃO

2023

- **REACT.JS**

React é uma biblioteca JavaScript, desenvolvida pelo Facebook em 2011, código aberto com foco na criação de interfaces gráficas em aplicações web. React é focado na otimização do desempenho server-side por meio de funções JavaScript denominadas de componentes, permitindo modularidade para a criação de páginas web.

COMO INSTALAR O REACT VIA NODE.JS:

1. **Instale o Node.js:** Baixe e instale o Node.js em seu sistema Windows. O Node.js inclui o npm (Node Package Manager) que é necessário para instalar o React.
2. **Crie um novo projeto React:** Abra o terminal ou prompt de comando e navegue até o diretório onde deseja criar o projeto. Execute o seguinte comando para criar um novo aplicativo React:

```
npx create-react-app mern-sign-up
```

1. **Navegue até o diretório do projeto:** Use o comando cd para navegar até o diretório do seu projeto.
2. **Inicie o servidor de desenvolvimento:** Execute o seguinte comando para iniciar o servidor de desenvolvimento do React:

```
npm start
```

Isso iniciará o servidor de desenvolvimento e abrirá o seu novo aplicativo React em seu navegador padrão.

COMO ESTRUTURAR UM PROJETO REACT:

A estrutura básica de um projeto React criado com o create-react-app é a seguinte:

```
nome-do-seu-app/
  README.md
```

```
node_modules/
package.json
.gitignore
public/
  index.html
  favicon.ico
src/
  App.css
  App.js
  App.test.js
  index.css
  index.js
  logo.svg
```

node_modules/: Este diretório contém todas as dependências do projeto.

package.json: Arquivo que mantém as informações do projeto e as dependências utilizadas.

public/: Contém o arquivo HTML principal e outros recursos estáticos.

src/: Contém o código fonte do aplicativo React. Aqui, você encontrará o componente principal, CSS e arquivos de teste.

IMPORTAÇÕES E ESTADOS INICIAIS:

```jsx
// App.js
import React, { useState } from 'react';
import './App.css';
import Dashboard from './Dashboard';
import Api from "./api.js"
const api = new Api();

function App() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const [users, setUsers] = useState([]);
  const [isLogin, setIsLogin] = useState(true);
  const [isLoggedIn, setIsLoggedIn] = useState(false);
```

O código importa o React e os Hooks `useState` para gerenciar o estado do componente. Também importa um componente `Dashboard` e um módulo Api. Já o componente App começa definindo seu estado inicial usando o Hook `useState`. Ele rastreia informações como nome, email, senha, uma lista de usuários, uma variável `isLogin` para alternar entre tela de login e registro, e uma variável isLoggedIn para controlar o estado de autenticação.

FUNÇÕES DE MANIPULAÇÃO DE LOGIN, REGISTRO E LOGOUT:

As funções `handleLogin` e `handleRegister` são responsáveis por fazer solicitações assíncronas para autenticar e registrar usuários, respectivamente. Elas fazem uso do módulo Api para se comunicar com o servidor, `handleLogout` é uma função simples que redefine o estado para desconectar o usuário.

```
async function handleLogin(){
try{
  const response = await api.loginUser({email, password});
  const user = response.data.name;

  if (user) {
    setName(user);
    setIsLoggedIn(true);
  }
}catch (e){
  setIsLoggedIn(false);
  alert("Usuário ou senha incorreto");
}
};

async function handleRegister(){
  try{
    if(!name || !email || !password){
      return alert("Preencha os campos!");
    }

    await api.registerUser({name, email, password});
    const newUser = { name, password };
    setUsers([...users, newUser]);
    setIsLoggedIn(true);
    alert('Registro realizado com sucesso!');
  } catch(e){
    setIsLoggedIn(false);
    alert("Usuário já cadastrado!");
  }
};

const handleLogout = () => {
  setIsLoggedIn(false);
  setName('');
  setPassword('');
};
```

ESTRUTURA DA PÁGINA DE LOGIN:

A função `render` retorna um componente condicional. Se o usuário estiver logado, o componente `Dashboard` é exibido. Caso contrário, são exibidos os componentes de login e registro, dependendo da variável `isLogin`. Logo, se `isLogin` for verdadeiro, este trecho renderiza um formulário de login com campos para o email e senha, com botões para submeter os dados ou alternar para a tela de registro. Por fim, se `isLogin` for falso, este trecho exibe um formulário de registro com campos para o nome de usuário, email e senha, com botões para submeter os dados ou alternar para a tela de login.

```
return (
  <div className="App">
    {isLoggedIn ? (
      <Dashboard name={name} handleLogout={handleLogout} />
    ) : (
      <div className="form-container">
        {isLogin ? (
          <div className="form">
            <h2>Login</h2>
            <input
              type="text"
              placeholder="Email"
              value={email}
              onChange={(e) => setEmail(e.target.value)}
            />
            <input
              type="password"
              placeholder="Senha"
              value={password}
              onChange={(e) => setPassword(e.target.value)}
            />
            <button onClick={handleLogin}>Entrar</button>
            <p>
              Não tem uma conta? <span onClick={() => setIsLogin(false)}>Registrar</span>
            </p>
          </div>
        ) : (
```

```
) : (
  <div className="form">
    <h2>Cadastro</h2>
    <input
      type="text"
      placeholder="Nome de usuário"
      value={name}
      onChange={(e) => setName(e.target.value)}
    />
    <input
      type="text"
      placeholder="Email"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
    />
    <input
      type="password"
      placeholder="Senha"
      value={password}
      onChange={(e) => setPassword(e.target.value)}
    />
    <button onClick={handleRegister}>Registrar</button>
    <p>
      Já tem uma conta? <span onClick={() => setIsLogin(true)}>Entrar</span>
    </p>
  </div>
```

- **MONGODB**

O MongoDB é um Banco de Dados Orientado a Documentos ou NoSQL, ou seja, ele é composto por registros que são estruturados de maneira mais *"informal"*, pois o mesmo não possui os conceitos de colunas, chaves estrangeiras entre outras formalidades que os Bancos de Dados Relacionais possuem. Então suas informações de cada registro ficam contidas nele mesmo ao invés de serem organizadas em tabelas, desta forma facilitando para trabalhar-se o crescimento dos dados.

Ele possui base de dados para separar o conjunto de informações, mas ao invés de tabelas ele possui collections (recebe esse nome por ser uma coleção de documentos, que como chama-se os registros no banco de dados), Ele também não existe a definição rígida de cada coluna dizendo que ela precisa ser um campo texto de um determinado tamanho, nele cada documento possui sua própria organização que pode ser diferente dos demais documentos da collection (não é aconselhável)

Ele possibilita a criação de Schemas que são conjuntos de definições das regras dos campos, seus conteúdo e até mesmo validação dos valores possíveis, desta forma é possível utilizá-lo de forma mais rígida se assim preferir.

- **COMO INSTALAR**

Aqui será mostrado os passos para que o MongoDB seja rodado de maneira local em sua versão Community. Primeiro seguinte instrução deverá ser executada em seu terminal:

```
docker run --name mongodb -d mongo:latest
```

Esse comando irá baixar em sua máquina a última versão disponível do MongoDB já configurada para rodar na porta 27017. Agora, o cliente deve ser acessado, para isso basta executar comando a seguir:

```
docker exec -it mongodb mongo
```

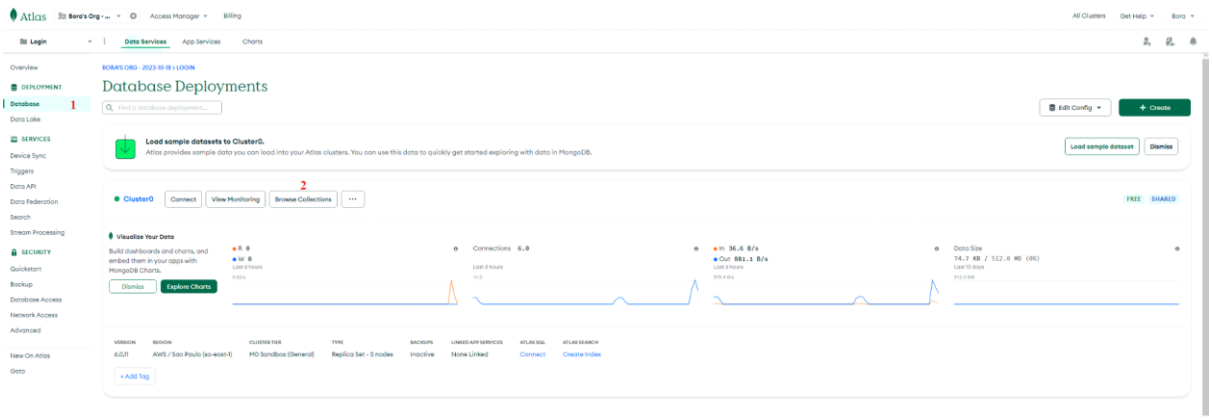
Com isso feito você já terá acesso ao MongoDB, mas caso prefira também é possível fazer esse processo de instalação com o auxílio de ferramentas de interface gráfica, como o Mongo Express.

- **COMANDOS**

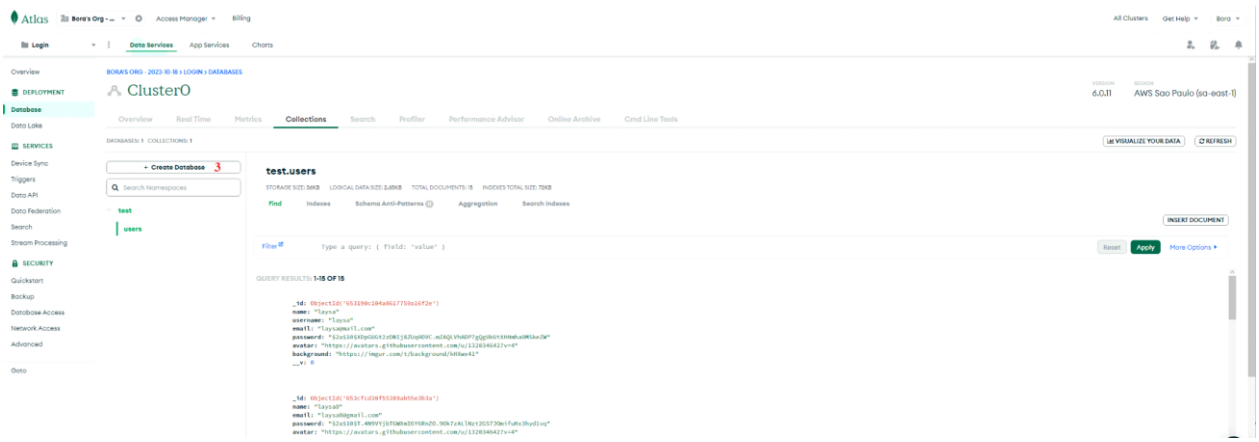
A melho forma de utilizar o MongoDB (principalmente em projetos pequenos) é usando o MongoDB Atlas, um serviço que permite o uso do banco de dados na nuvem, o que facilita bastante o funcionamento da ferramenta.

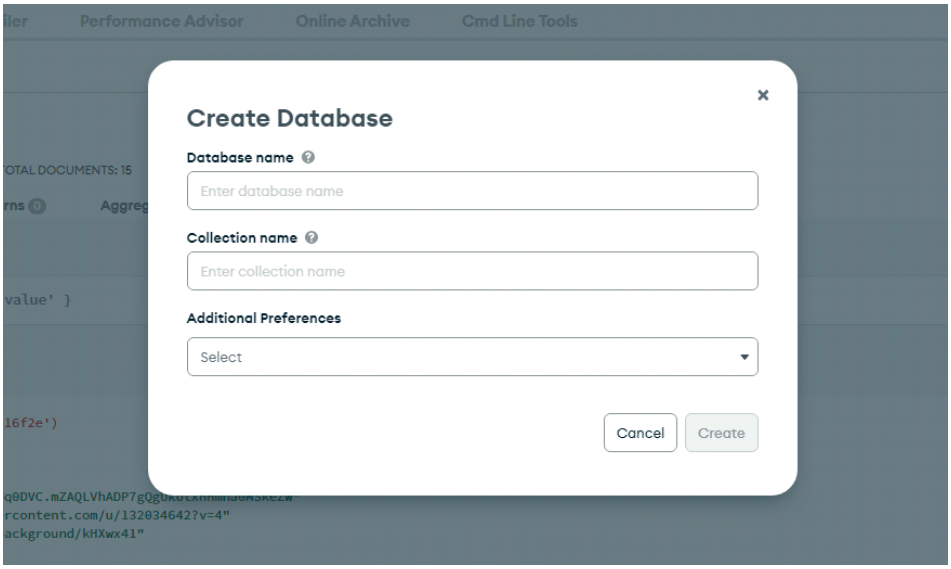
Para isso é necessário criar uma conta no MongoDB Atlas e fazer algumas configurações de segurança caso seja necessário. Alguns dos principais comandos são:

Criar coleções: Para criar as coleções basta ir no lado esquerdo, em “Database” e, na parte do cluster ir em “Browse Collections” (Os botões serão numerados com números vermelhos para facilitar a visualização)

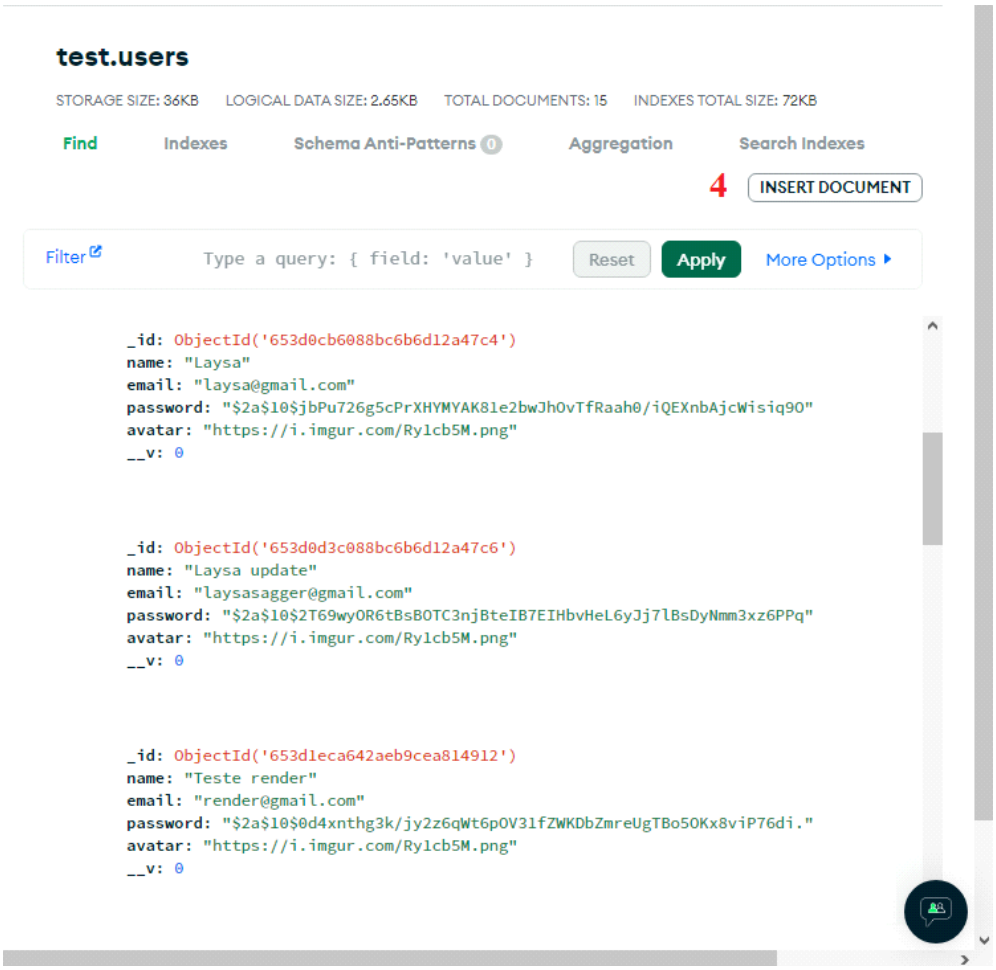


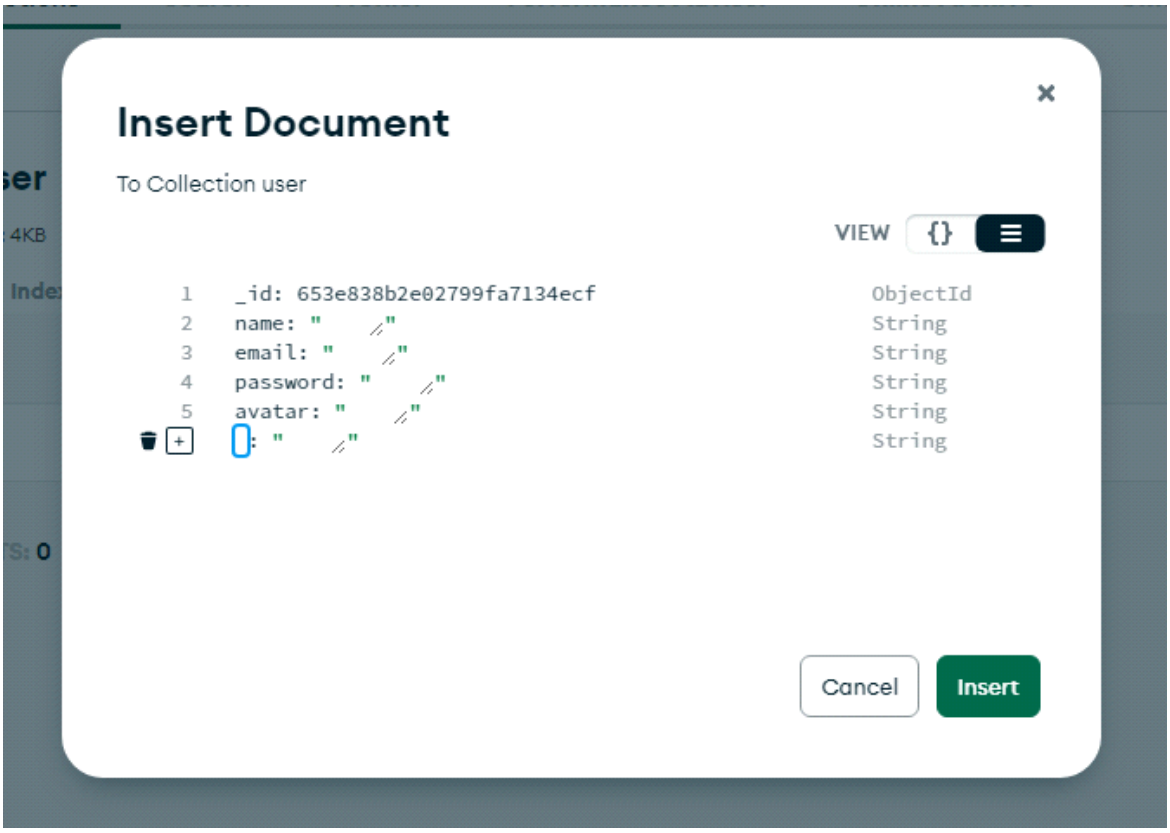
Após isso, para criar uma Database, basta ir na partição “Collections” (que provavelmente terá algumas databases padrão do Atlas, que poderão ser excluídas) e clicar no botão “ + Create Database”, onde você poderá escolher o nome da Database, o nome da primeira coleção e o tipo dela.





Criar documentos: Os documentos são as informações de cada usuário, contendo: nome, email, senha, avatar e um ID que é gerado automaticamente toda vez que um novo registro é inserido. Para isso, é necessário clicar em “INSERT DOCUMENT”


test.users
STORAGE SIZE: 36KB
LOGICAL DATA SIZE: 2.65KB
TOTAL DOCUMENTS: 15
INDEXES TOTAL SIZE: 72KB
Find
Indexes
Schema Anti-Patterns 0
Aggregation
Search Indexes
4
INSERT DOCUMENT
Filter
Type a query: { field: 'value' }
Reset
Apply
More Options
_id: ObjectId('653d0cb6088bc6b6d12a47c4')
name: "Laysa"
email: "laysa@gmail.com"
password: "$2a$10$jbPu726g5cPrXHYMYAK81e2bwJhOvTfRaah0/iQEXnbAjcWisiq9O"
avatar: "https://i.imgur.com/Ry1cb5M.png"
__v: 0
_id: ObjectId('653d0d3c088bc6b6d12a47c6')
name: "Laysa update"
email: "laysasagger@gmail.com"
password: "$2a$10$2T69wyOR6tBsBOTC3njBteIB7EIHbvHeL6yJj7lBsDyNmm3xz6PPq"
avatar: "https://i.imgur.com/Ry1cb5M.png"
__v: 0
_id: ObjectId('653d1eca642aeb9cea814912')
name: "Teste render"
email: "render@gmail.com"
password: "$2a$10$0d4xnthg3k/jy2z6qWt6pOV31fZWKDbZmreUgTBo5OKx8viP76di."
avatar: "https://i.imgur.com/Ry1cb5M.png"
__v: 0



Esse é um exemplo de um campo vazio de documento. Na parte da esquerda, antes dos dois pontos, coloca-se o campo que será preenchido, após os dois pontos, entre parênteses, o valor que definirá o dado (se o dado for um number os parênteses não são necessários).

Filtrar informações: Muitas vezes será necessário achar alguns dados específicos no banco de dados, porém com a quantidade avassaladora de informações pode ser muito complicado, por este motivo existe o campo “Filter”, onde você deverá digitar qual e campo e o valor de interesse para assim efetuar a busca.

test.users

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 2.65KB TOTAL DOCUMENTS: 15 INDEXES TOTAL SIZE: 72KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

INSERT DOCUMENT

Filter 5 Type a query: { field: 'value' } Reset Apply More Options

QUERY RESULTS: 1-15 OF 15

```
_id: ObjectId('653190c104a8617759a16f2e')
name: "laysa"
username: "laysa"
email: "laysa@mail.com"
password: "$2a$10$XDpGUGt2zDNIj8ZUq0DVC.mZAQLVhADP7gQgUkUtXHHmha0MSkeZW"
avatar: "https://avatars.githubusercontent.com/u/132034642?v=4"
background: "https://imgur.com/t/background/kHXwx41"
__v: 0


_id: ObjectId('653cfcd30f55389ab55e3b3a')
name: "laysa8"
email: "laysa8@gmail.com"
password: "$2a$10$T.4N9VYjbTGWXmI6Y6RnZO.9Ok7zALlNzt2GS7JOmifuHx3hyd1vq"
avatar: "https://avatars.githubusercontent.com/u/132034642?v=4"
__v: 0


_id: ObjectId('653d0cb6088bc6b6d12a47c4')
name: "Laysa"
email: "laysa@gmail.com"
```

test.users

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 2.65KB TOTAL DOCUMENTS: 15 INDEXES TOTAL SIZE: 72KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

INSERT DOCUMENT

Filter {name: 'miguel'} Reset Apply More Options

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('653d35824067774ed8043b3f')
name: "miguel"
email: "miguel@gmail.com"
password: "$2a$10$fevIgxfPwNE6fu3S8d7eauxPHMecjPTJrTSdKpOI9KoWnqNksaXT6"
avatar: "https://i.imgur.com/Ry1cb5M.png"
__v: 0
```

- **QUE É O NODE.JS?**

Node.js é uma plataforma de desenvolvimento de software de código aberto que permite a criação de uma ampla variedade de aplicativos, incluindo servidores web altamente escaláveis e eficientes, bem como aplicativos de linha de comando, ferramentas de automação e aplicativos de desktop. Ele utiliza o motor JavaScript V8 da Google para executar código JavaScript no lado do servidor (back-end), em vez de no navegador.

Ao instalar o Node.js, o usuário também obtém acesso ao Gerenciador de Pacotes do Node (NPM, na sigla em inglês). Que é um dos pilares do Node.js é oferece acesso a uma vasta coleção de pacotes reutilizáveis. O NPM simplifica o gerenciamento de dependências, tornando mais fácil a integração de módulos e bibliotecas externas. Além disso, ele pode ser usado para automatizar tarefas de compilação, economizando tempo e esforço no desenvolvimento.

- COMO INSTALAR O NODE.JS:

- Acesse o site oficial do Node.js em https://nodejs.org/en/download
- Baixe o instalador da versão LTS (recomendada para estabilidade).
- Execute o instalador e siga o assistente de instalação.
- Aceite os termos de licença e mantenha as opções padrão.
- Clique em "Next" até concluir a instalação.

Depois de concluir a instalação do Node.js, é aconselhável verificar se tudo foi instalado corretamente. Para fazer isso, basta abrir o terminal ou prompt de comando e digitar "`node -v`" e "`npm -v`". Se esses comandos retornarem versões, significa que o Node.js foi instalado com sucesso.

- ONDE O NODE.JS SERÁ USADO NESTE PROJETO?

Nesta aplicação MERN stack (MongoDB, Express, React, Node.js), o Node.js atuará como o servidor, sendo essencial para a página de login do usuário. Ele será responsável por lidar

com a autenticação, a comunicação com o banco de dados MongoDB e a gestão das solicitações e respostas.

O Node.js utiliza o framework Express.js para configurar o servidor, conecta-se ao MongoDB com o Mongoose e mantém a segurança através de práticas como o armazenamento seguro de senhas. De forma prática o Node.js responde apropriadamente às solicitações HTTP, garantindo uma experiência segura de login para os usuários React.

- **EXPRESS**

Express, é um framework de aplicativo web para Node.js, construído para facilitar o desenvolvimento de aplicativos web e APIs robustas. É minimalista e flexível, permitindo que os desenvolvedores construam aplicativos web de forma eficiente. Express fornece um conjunto de funcionalidades essenciais para lidar com solicitações HTTP, roteamento, middlewares e muito mais.

- COMO INSTALAR O EXPRESS:
- Certifique-se de ter o Node.js instalado no seu sistema.
- Em seu diretório de projeto, abra o terminal ou prompt de comando.
- Execute `npm init -y` que criará um arquivo `package.json`.
- Execute `npm install express –save` comando que instalará o express no projeto.

- O QUE É O PACKAGE.JSON:

O package.json é um arquivo fundamental em um projeto Node.js. Ele atua como um registro que lista todas as informações sobre o projeto, como suas dependências (bibliotecas usadas no projeto), metadados (nome do projeto, versão, descrição) e scripts (comandos personalizados para execução).

Basicamente, o package.json é como um documento de identidade do seu projeto Node.js, informando quais recursos ele precisa e como ele deve funcionar. Isso é importante para gerenciar e compartilhar projetos, garantindo que outros desenvolvedores possam reproduzi-lo corretamente.

Quando se executa o comando `npm init -y` o arquivo package.json e instalado de forma automática então o usuário pode alterar as informações de metadados e scripts se necessário.

- ONDE O EXPRESS SERÁ UTILIZADO:

Express.js sera utilizado devido à sua capacidade de simplificar o desenvolvimento de rotas, gerenciar middlewares, integrar-se perfeitamente ao aplicativo e oferecer uma experiência de desenvolvimento eficiente e segura. O que contribui para a criação de uma página de login funcional e protegida.

- **DESENVOLVIMENTO DO BACK-END:**

No contexto de uma API de uma página de login o back-end é responsável por gerenciar o processo de autenticação e autorização dos usuários. Isso envolve a validação das credenciais fornecidas, a comunicação com um banco de dados para verificar a existência do usuário. Além disso, o back-end lida com a segurança, garantindo que as informações de login sejam transmitidas de forma segura por meio de protocolos criptografados, como HTTPS.

A estrutura back-end do projeto, localizada na pasta “backend” que possui um diretório "src", que inclui pastas como "controllers" para a lógica das rotas, "middlewares" para processar solicitações, "routes" para definições de rotas, e "services" para lógica de negócios.

Há também arquivos essenciais como "db.js" para conexão com o banco de dados e "models.js" para esquemas de dados também estão presentes. Já o arquivo "index.js" atua como o ponto de entrada do servidor Express, e os arquivos "package-lock.json" e

"package.json" gerenciam as dependências. Essa organização garante uma estrutura clara e eficaz para o desenvolvimento e manutenção do projeto.

- DEPENDÊNCIAS:

A utilização de bibliotecas e dependências é uma prática essencial que permite aos desenvolvedores acelerarem o processo de criação de aplicativos e sistemas. As dependências são pacotes de código pré-desenvolvido que fornecem funcionalidades específicas, recursos e soluções para problemas comuns. Nesse projeto foram usadas as seguintes bibliotecas no desenvolvimento do back-end.

- BCRYPTJS:

O bcryptjs é uma biblioteca JavaScript que oferece funções para criptografar senhas de forma segura. Seu principal objetivo é realizar o processo de hashing de senhas. O hashing é um método matemático que converte uma senha em uma sequência de caracteres ilegível. A ideia é que o resultado do hashing seja irreversível, o que significa que é praticamente impossível recuperar a senha original a partir do hash. Nesse projeto o bcrypt foi utilizado para criptografar a senha digitada pelo usuário

Para instalar o bcryptjs, entre no diretório do projeto e digite o seguinte comando no terminal:

```
npm install bcryptjs.
```

Após esse comando, verifique se o mesmo consta no seu arquivo package.json, na seção de "dependencies", onde deverá conter o bcryptjs e sua versão, por exemplo "bcryptjs": "^2.4.3".

- CORS:

Cors (Cross-Origin Resource Sharing) é um mecanismo crítico para garantir que as solicitações de origem cruzada sejam tratadas de maneira segura e que as respostas sejam

restritas a origens confiáveis. É uma consideração fundamental ao criar aplicativos web que interagem com diferentes domínios e desempenha um papel vital na proteção dos usuários e na segurança das aplicações.

Para instalar o cors, entre no diretório do projeto e digite o seguinte comando no terminal:

```
npm install cors
```

Nesta aplicação, empregou-se a utilização do pacote CORS devido à necessidade de permitir a comunicação entre a aplicação front-end e o back-end, os quais estavam operando em portas distintas. Com o intuito de resolver problemas de requisições com origens cruzadas, implementou-se o CORS. Inicialmente, o pacote CORS foi importado na aplicação e, em seguida, configurou-se de forma a permitir todas as origens. Esse procedimento possibilitou a interação adequada entre as partes, solucionando questões relacionadas à segurança e permissões.

- DOTENV:

O dotenv é uma biblioteca que permite carregar variáveis de ambiente a partir de um arquivo .env. Ele é útil para armazenar configurações sensíveis ou valores de configuração em um local seguro. O uso de variáveis de ambiente com dotenv é uma prática segura para armazenar informações confidenciais, como chaves de API, senhas de banco de dados e outras configurações sensíveis, sem expô-las em seu código-fonte.

Para instalar o dotenv, entre no diretório do projeto e digite o seguinte comando no terminal:

```
npm install dotenv
```

Em nosso código, implementamos o uso do dotenv para configurar a porta padrão na qual o aplicativo será executado, estabelecendo a porta 3001 como valor padrão, a menos que a

variável de ambiente PORT seja explicitamente definida. Isso garante que, se o servidor onde hospedamos nossa aplicação back-end estiver configurado para utilizar uma porta diferente, nossa aplicação ainda funcionará de maneira adequada e sem problemas.

- MONGOOSE:

O é uma biblioteca de modelagem de objetos MongoDB para Node.js. Ele fornece uma camada de abstração para interagir com bancos de dados MongoDB. Ela simplifica a interação com bancos de dados MongoDB, permitindo definir modelos de dados, realizar consultas e validações de forma mais eficiente.

Para instalar o Mongoose, entre no diretório do projeto e digite o seguinte comando no terminal:

```
npm install mongoose
```

Na nossa aplicação, empregamos o Mongoose para estabelecer a conexão com o banco de dados e para definir a estrutura dos documentos no banco. O Mongoose facilita a definição e organização dos dados no banco de dados, tornando o desenvolvimento mais eficiente e organizado.

- NODEMON:

O Nodemon é uma ferramenta que monitora alterações em arquivos no seu aplicativo Node.js e reinicia automaticamente o servidor quando uma alteração é detectada. O Nodemon é útil durante o desenvolvimento para evitar a necessidade de reiniciar manualmente o servidor a cada modificação no código, e portando foi utilizado nesse projeto para facilitar o desenvolvimento.

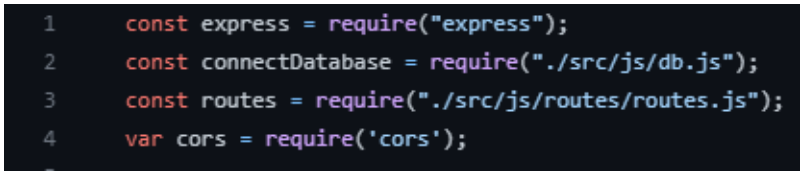
Para instalar o Nodemon, entre no diretório do projeto e digite o seguinte comando no terminal:

```
npm install --save-dev nodemon
```

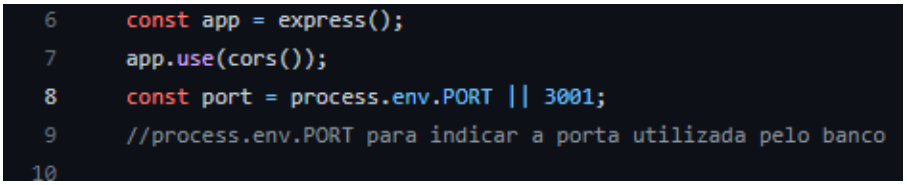
- **EXPLICAÇÃO DO DESENVOLVIMENTO DO PROJETO:**

- INDEX.JS

O arquivo index.js atua como o ponto de entrada do servidor Express. É possui a seguinte estrutura. Logo nas primeiras linhas, como mostra na imagem abaixo, estão as importações dos módulos necessários.



Logo em seguida, está a criação de uma intância de Express. O app é o ponto central para o roteamento e o tratamento de solicitações no servidor Express.



Na linha 7 foi usado o middleware ‘app.use(cors())` que é uma configuração que permite que um servidor web compartilhe recursos com origens diferentes, garantindo que o navegador permita solicitações entre domínios separados. Isso é essencial para a comunicação segura entre o front-end e o back-end de uma aplicação web.

Na linha 8, definimos a variável "port" que especificará a porta em que nosso servidor estará ativo. A lógica por trás disso é a seguinte: se a variável de ambiente "process.env.PORT" estiver definida (o que é comum em ambientes de hospedagem na nuvem), o servidor utilizará essa porta para a sua execução.

Caso essa variável não esteja definida, o servidor usará a porta 3001 como alternativa. Essa abordagem oferece flexibilidade ao permitir que o ambiente de hospedagem defina a porta, enquanto fornece uma porta padrão caso isso não ocorra.

```
11    connectDatabase();
12
13    app.use(express.json()); // Comando para permitir que a aplicação receba JSON
14    app.use(routes);
15
16    app.listen(port, () => console.log(`Servidor rodando na porta ${port}`));
```

Na linha 11, foi chamado a função connectDatabase() para estabelecer a conexão com o banco de dados. Função que está definida no arquivo bd.js.

A linha 13 possui o app.use(express.json()); que é um middleware que permite que a aplicação receba dados no formato JSON em solicitações HTTP. Isso é essencial para processar solicitações e respostas no formato JSON.

Já na linha 12 foi adicionado as rotas definidas no módulo routes à nossa aplicação. Isso permite que o Express encaminhe solicitações HTTP para as rotas corretas.

Por fim, na linha 16, foi iniciado o servidor na porta especificada. Quando o servidor é iniciado com sucesso, ele exibe uma mensagem indicando a porta em que está rodando.

- DB.JS

O arquivo db.js foi desenvolvido para conter as configurações para conexão com o banco de dados.

```
1    const mongoose = require("mongoose");
2    // Conectando ao Mongo.db
```

Neste passo, foi importado a biblioteca "mongoose" para permitir a conexão ao banco de dados MongoDB.

```
3  const connectDatabase = () => {
4      console.log("Wait connecting to the databse");
5
6      mongoose.connect(
7          "mongodb+srv://login:login123@cluster0.m6gz8me.mongodb.net/", { useNewUrlParser: true, useUnifiedTopology: true }
8
9      ).then(() => console.log("MongoDB Atlas Connected")).catch((error) => console.log(error));
10 };
11
12 module.exports = connectDatabase;
```

Na linha 3 foi declarado a função connectDatabase, que será responsável por estabelecer a conexão com o banco de dados. Ela começa mostrando uma mensagem no console para indicar que está tentando estabelecer a conexão.

Nas linha entre 6 e 10, foi utilizado o método mongoose.connect() para efetuar a conexão com o banco de dados MongoDB. A URL de conexão está definida como o primeiro argumento da função connect. As opções { useNewUrlParser: true, useUnifiedTopology: true } são usadas para configurar a conexão..then() é usado para lidar com o sucesso da operação e exibir uma mensagem de confirmação no console..catch() lida com quaisquer erros que possam ocorrer durante a conexão e os exibe no console.

E por fim é exportado a funação connectDatabase para que ela possa ser usada em outras partes do aplicativo.

- MODELS.JS

O arquivo models.js foi usado para criar esquemas de dados que serão implementados no MongoDB

```
1  //Models é a forma de bolo de um usuário
2  const mongoose = require('mongoose'); //Biblioteca Node.js
3  const bcrypt = require("bcryptjs"); //Criptografar as senhas
```

Nas primeiras linhas é importado o módulo Mongoose, que é usado para definir esquemas (schemas) e interagir com o MongoDB. É o módulo bcryptjs para criptografar senhas de usuário.

```
7   const UserSchema = new mongoose.Schema({
8       name:{
9           type: String,
10          required: true,
11      },
12      email: {
13          type: String,
14          required: true,
15          unique: true , //para cada email ser unico
16          lowercase:true,
17      },
18      password: {
19          type: String,
20          required: true,
21          select: false,
22      }
23  })
```

Entre as linhas 7 e 23 contém o código que define o esquema de usuário (UserSchema) usando a função mongoose.Schema().

O esquema especifica a estrutura dos documentos de usuário, incluindo os campos name, email e password. Cada campo tem um tipo de dados e pode ter várias opções, como required e unique. O campo select: false impede que a senha seja selecionada por padrão em consultas, fornecendo segurança adicional.

```
25  //Criptografia:
26  //Esse 10 indica que var ter 10 saltos de criptografia
27  UserSchema.pre("save", async function (next) {
28      this.password = await bcrypt.hash(this.password, 10);
29      next();
30  });
```

O código entre as linhas 26 e 30 contém um middleware "pre" do Mongoose que é executado antes de salvar um documento no banco de dados.O middleware utiliza o módulo bcrypt para criptografar a senha do usuário antes de salvá-la no banco de dados. O valor 10 representa o custo computacional da criptografia, tornando-a mais segura.

```
32  const User = mongoose.model("User", UserSchema);
33
34  module.exports = User;
```

Por fim foi criado criamos o modelo de usuário (User Model) usando a função mongoose.model(). O modelo define como os dados dos usuários são armazenados e recuperados no banco de dados.

E a última linha apresenta comando para exportar o modelo de usuário para que ele possa ser usado em outras partes do aplicativo.

- ROUTES/ROUTE.JS

A pasta "routes" do projeto contém as definições de rotas, determinando como as URLs se relacionam com os controladores. Esse arquivo route.js é o arquivo que associa as rotas dos caminhos específicos, que são user e auth.

```
1   const {Router} = require("express");
2
3   // Importando rotas
4   const userRouter = require("./user.routes");
5   const authRouter = require("./auth.routes");
6
7   const router = Router();
8   router.use("/users", userRouter);
9   router.use("/auth", authRouter);
10
11  module.exports = router;
```

Neste código, primeiramente, uma instância do roteador Express é criada, o que permite definir rotas para manipular solicitações HTTP. As rotas específicas estão localizadas nos arquivos "user.routes" e "auth.routes", que são importados na segunda e terceira linhas.

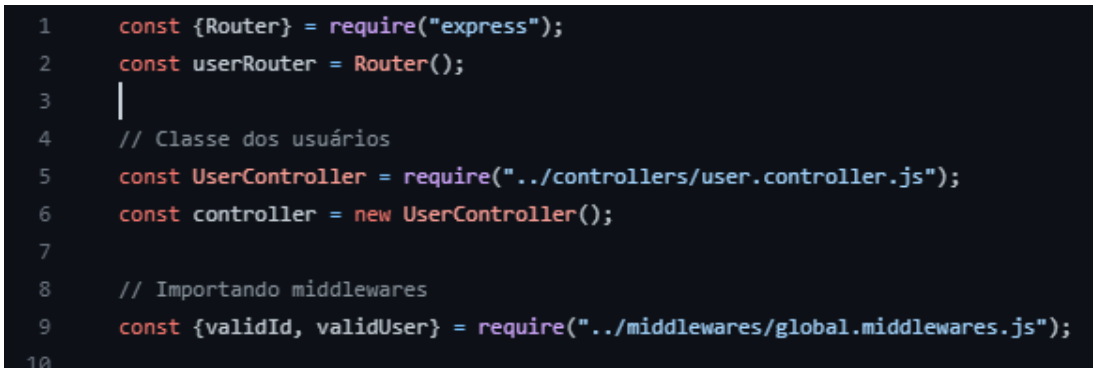
Em seguida, um roteador é instanciado usando a variável "Router" importada do módulo "express". Isso permite a criação de rotas para diferentes caminhos.

Posteriormente, as rotas importadas, "userRouter" e "authRouter", são associadas a caminhos específicos no roteador. A função "router.use()" é utilizada para especificar que as rotas definidas em "userRouter" devem ser acessadas sob o caminho "/users", enquanto as rotas de "authRouter" são acessadas sob o caminho "/auth".

Por fim, o roteador é exportado, tornando-o disponível para ser utilizado em outros lugares da aplicação, para que as solicitações feitas aos caminhos "/users" e "/auth" sejam devidamente direcionadas para os controladores apropriados.

- ROUTES/USER.ROUTE.JS

Este código define e exporta rotas para manipular operações relacionadas a usuários.



Primeiro importado e depois criado uma instância do roteador Express e é criada e atribuída a "userRouter". Isso permite definir rotas para manipular solicitações HTTP relacionadas aos usuários.

Em seguida, a classe "UserController" é importada e uma instância dessa classe é criada com "controller". O "UserController" lida com as operações relacionadas aos usuários.

Também são importados middlewares, como "validId" e "validUser", que serão aplicados a rotas específicas para validação de identificação e de usuário.validId e validUser: Importa middlewares que serão executados antes das rotas para validar dados e autenticar o usuário.

```
11    // Funções de Usuário
12    userRouter.post("/", controller.create);
13    userRouter.get("/", controller.findAll);
14    userRouter.get("/:id", validId, validUser, controller.findById);
15
16    module.exports = userRouter;
```

Nas linhas seguintes são definidas três rotas:

- Uma rota POST ("/") que chama o método "create" no controlador, responsável por criar um novo usuário.
- Uma rota GET ("/") que chama o método "findAll" no controlador, utilizado para buscar todos os usuários.
- Uma rota GET ("/:id") que chama os middlewares "validId" e "validUser" antes de executar o método "findById" no controlador. Essa rota é usada para buscar um usuário com um ID específico, com validações adicionais aplicadas.

Por fim, "userRouter" é exportado, tornando-o disponível para ser utilizado em outros lugares da aplicação.

- ROUTES/AUTH.ROUTE.JS

Este arquivo define e exporta rotas específicas para lidar com operações de autenticação, incluindo o processo de login de usuários.

```
1     const {Router} = require("express");
2     const authRouter = Router();
3
4     //Classe de autentificação
5     const AuthController = require("../controllers/auth.controller.js");
6     const auth = new AuthController();
7
8     // Funções de Autentificação
9     authRouter.post("/", auth.login);
10
11    module.exports = authRouter;
```

As primeiras linhas cóntem, uma instância do roteador Express é criada e atribuída a "authRouter". Essa instância permite a definição de rotas para lidar com operações de autenticação.

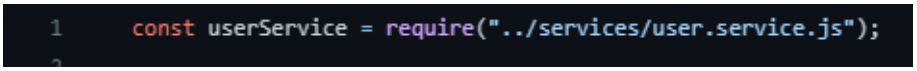
A classe "AuthController" é importada e uma instância dessa classe é criada como "auth". Essa classe lida com operações de autenticação, como o login de usuários.

É definida uma única rota uma rota POST ("/") que chama o método "login" no controlador "auth", utilizado para autenticar um usuário.

Por fim, "authRouter" é exportado, possibilitando que esteja disponível para ser utilizado em outros lugares.

- CONTROLLER/USER.CONTROLLER.JS

Este código define um controlador "UserController" que lida com operações de criação, busca e recuperação de informações de usuários.



Na primeira linha é importado o userService que é responsável responsável por executar operações relacionadas a usuários.

```
3  class UserController {
4      //Função de callback
5      //async para dizer que é uma função assincrona
6      create = async (req, res) => {
7          try {//constante que verifica todos os campos
8              const { name, email, password} = req.body;
9
10             if (!name || !email || !password) {
11                 res.status(400).send({ message: "Preencha todos os espaços" });
12             }
13
14             //await é usado junto com async
15             const user = await userService.createService(req.body);
16
17             if (!user) {
18                 return res.status(400).send({ message: "Erro ao criar usuário" });
19             }
20
21             res.status(201).send({
22                 message: "Usuário criado com sucesso!",
23                 user: {
24                     id: user._id,
25                     name,
26                     email,
27                 }
28             });
29         } catch (err) {
30             res.status(500).send({ message: err.mensage });
31         }
32     }
```

Na linha 3 é criado uma classe chamada UserController que conterá as funções do controlador.

Entre as linhas 6 e 32 está o método "create" que lida com a criação de um novo usuário. Ele verifica se todos os campos necessários, como nome, email e senha, foram fornecidos. Se algum campo estiver ausente, retorna um status 400 (Bad Request). Caso contrário, chama um serviço para criar o usuário e, se bem-sucedido, retorna um status 201 (Created) com uma mensagem de sucesso e os detalhes do usuário..

```
34        findAll = async (req, res) => {
35            try {
36                const users = await userService.findAllService();
37
38                if (users.length === 0) {
39                    return res.status(400).send({ message: "There are no registered users" });
40                }
41                res.send(users);
42            } catch (err) {
43                res.status(500).send({ message: err.message })
44            }
45        };
```

O método "findAll" busca e retorna todos os usuários registrados. Se não houver usuários, retorna um status 400 com uma mensagem informativa.

```
46
47        findById = async (req, res) => {
48            try {
49                const user = req.user;
50                res.send(user);
51            } catch (err) {
52                res.status(500).send({ message: err.mensage })
53            }
54        }
55    }
56
57    module.exports = UserController;
```

O método "findById" retorna as informações de um usuário com base no ID fornecido na solicitação.Por fim, exportamos a classe UserController para que ela possa ser usada em outras partes do aplicativo.

- CONTROLLER/AUTH.CONTROLLER.JS

Este arquivo define um controlador "AuthController" que lida com a autenticação de usuários, verificando as credenciais fornecidas (email e senha).

```
1    // Autentificar o Usuário
2    const bcrypt = require("bcryptjs");
3    const loginService = require("../services/auth.service.js");
4
```

Nas primeiras linhas são importados os módulos bcryptjs, que é usado para verificar a senha do usuário.e o módulo loginService: Importa o serviço loginService que lida com a autenticação de usuário.

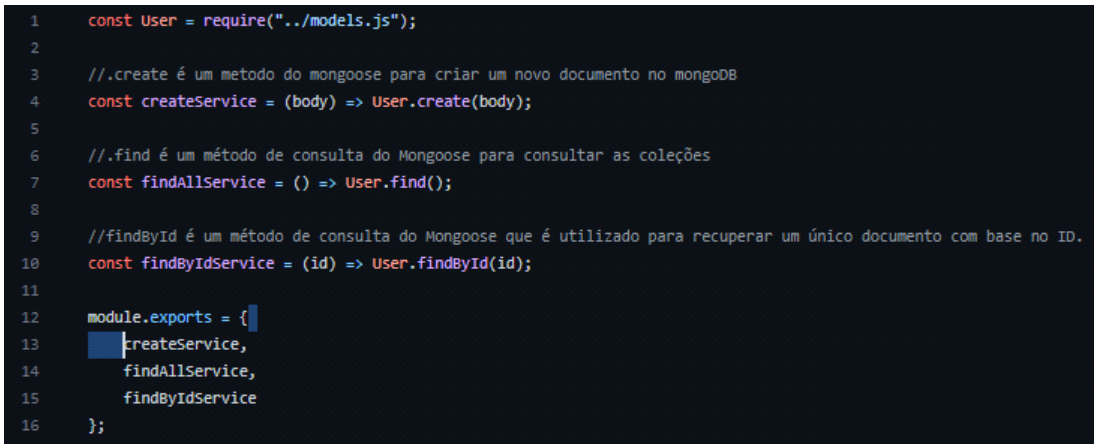
```
5  class AuthController{
6      login = async (req, res) => {
7          const {email, password} = req.body;
8          try{
9
10             const user = await loginService(email);
11             const passwordIsValid = await bcrypt.compare(password, user.password);
12
13             if(!user){
14                 return res.status(404).send({message: "Usuário ou senha não incorreto"});
15             }
16
17             if(!passwordIsValid){
18                 return res.status(400).send({message: "Usuário ou senha incorreto"});
19             }
20
21             res.status(200).send(user);
22         } catch(err){
23             res.status(500).send(err.message);
24         };
25
26     }
27 }
28
29 module.exports = AuthController
```

Esse arquivo possui um único método que é o método "login" que lida com a autenticação. Ele recebe o email e a senha fornecidos na solicitação, verifica se o usuário existe através do serviço "loginService" e, em seguida, compara a senha fornecida com a senha armazenada no banco de dados usando o "bcrypt.compare." Se o usuário não existe, retorna um status 404 (Not Found), se a senha não é válida, retorna um status 400 (Bad Request), e se a autenticação é bem-sucedida, retorna um status 200 (OK) com os detalhes do usuário

Por fim, exportamos a classe AuthController para que ela possa ser usada em outras partes do aplicativo.

- SERVICE/USER.SERVICE.JS

Este arquivo define serviços que interagem com o MongoDB usando o Mongoose para criar, buscar todos e buscar por ID documentos relacionados a usuários.



```
const User = require("../models.js");

//.create é um metodo do mongoose para criar um novo documento no mongoDB
const createService = (body) => User.create(body);

//.find é um método de consulta do Mongoose para consultar as coleções
const findAllService = () => User.find();

//findById é um método de consulta do Mongoose que é utilizado para recuperar um único documento com base no ID.
const findByIdService = (id) => User.findById(id);

module.exports = {
    createService,
    findAllService,
    findByIdService
};
```

Primeiro, importamos o modelo de usuário, que representa a estrutura dos documentos de usuário no banco de dados. O serviço "createService" utiliza o método ".create" do Mongoose para criar um novo documento no MongoDB com base nos dados fornecidos no parâmetro "body".

O serviço "findAllService" utiliza o método ".find" do Mongoose para consultar a coleção de documentos no MongoDB e retorna todos os registros presentes. O serviço "findByIdService" utiliza o método ".findById" do Mongoose para recuperar um único documento com base no ID fornecido como parâmetro. Esses serviços abstraem as operações de banco de dados relacionadas à criação, consulta e recuperação de informações de usuários no MongoDB .

- SERVICE/AUTH.SERVICE.JS

Neste código, temos um serviço que lida com a comunicação entre a aplicação e o banco de dados, usando o Mongoose.

```
1   //Responsável pela comunicação com o banco
2   //Veremos se o email passado por parâmetro confere dentro do banco
3   const User = require("../models.js");
4
5   const loginService = (email) => User.findOne({email: email}).select("+password");
6
7   module.exports = loginService;
```

O serviço "loginService" recebe um email como parâmetro e utiliza o método ".findOne" do Mongoose para procurar um documento na coleção de usuários onde o campo "email" corresponde ao email fornecido. O ".select("+password") é usado para incluir o campo de senha na busca, pois, normalmente, ele é ocultado por questões de segurança..

- MIDDLEWARES/GLOBAL.MIDDLEWARES.JS

Este código define funções de validação que são executadas antes dos controladores, garantindo a validade das solicitações, como verificar a validade do ID e a existência do usuário.

```
1   const mongoose = require("mongoose");
2   const userService = require("../services/user.service.js");
```

E importado o módulo Mongoose, que é usado para verificar IDs válidos. E o módulo userService que lida com operações de usuário.

```
const validId = (req, res, next) => {
    try {
        const id = req.params.id;

        //esse codigo é um padrão do mongoose para testar id se é valido
        if (!mongoose.Types.ObjectId.isValid(id)) {
            return res.status(400).send({ message: "Invalid ID" });
        }
        next();
    } catch (err) {
        res.status(500).send({ message: err.mensage })
    }
};
```

O middleware validId A função "validId" verifica se o ID fornecido nos parâmetros da solicitação é um ID válido, de acordo com os padrões do Mongoose. Caso contrário, retorna um status 400 (Bad Request).Se o ID não for válido, ele envia uma resposta de erro com a mensagem "ID inválido".

```
const validUser = async (req, res, next) => {
    try {
        const id = req.params.id;

        const user = await userService.findByIdService(id);

        if (!user) {
            return res.status(400).send({ message: "User not found" });
        }

        req.id = id;
        req.user = user;
        next();
    } catch (err) {
        res.status(500).send({ message: err.mensage })
    }
};

module.exports = {
    validId,
    validUser
}
```

O middleware validUser contém a função "validUser" que é assíncrona e valida se um usuário com o ID fornecido existe, consultando o serviço "findByIdService" do usuário. Se o usuário não for encontrado, retorna um status 400 (Bad Request). Caso contrário, armazena o ID e os detalhes do usuário na solicitação para uso posterioré usado para validar a existência de um usuário no banco de dados com base no ID fornecido.

Por fim, exportamos os middlewares em um objeto para que possam ser usados em outras partes do aplicativo.

- **ADICIONANDO A API BO SERVIDOR**

Após o desenvolvimento da parte de back-end da aplicação MERN (MongoDB, Express.js, React.js e Node.js), é fundamental concluir e implantar essa seção em um servidor para que a aplicação seja acessível na web. Neste projeto utilizamos o servidor Render.

Entretanto, cabe ressaltar que são boas práticas antes da implantação, assegurar que todas as configurações estejam corretas e que o back-end esteja funcionando conforme o previsto. Para isso, recomenda-se fazer testes para verificar o funcionamento.

Para esses testes podem ser usadas por exemplo a extensão do vscode Thunder Client ou o aplicativo Insomnia que ambas são ferramentas de cliente HTTP, projetadas para auxiliar desenvolvedores a testar e depurar APIs. Eles permitem fazer solicitações HTTP para endpoints de API, observar respostas, gerenciar cabeçalhos e parâmetros, além de automatizar e organizar testes de API.Com a aplicação testada e funcionando adequadamente , poderá ser feito o deploy no servidor sem maiores problemas.

Render é uma plataforma de hospedagem que simplifica a implantação de aplicativos da web, incluindo aqueles desenvolvidos com a pilha MERN. Os passos básicos para implantar o back-end no servidor Render são os seguintes:

- Criação de Conta Render: Caso ainda não tenha uma conta, é necessário criar uma em https://render.com.

- Criação de um Novo Serviço:Acessar a conta Render e clicar em "Adicionar um Novo Serviço" no painel de controle. Selecionar "Web Service" como tipo de serviço.
- Configuração do Serviço: Escolher um nome para o serviço. Selecionar o repositório do back-end no GitHub. Configurar as variáveis de ambiente necessárias, como a chave de conexão com o banco de dados.
- Implantação Automática: Configurar a implantação automática, se desejado, para que o back-end seja atualizado sempre que houver modificações no repositório.
- Implantação do Serviço: Clicar em "Criar Serviço" ou "Implantar Agora" para iniciar a implantação.
- Acesso ao Back-End: Após a conclusão da implantação, o servidor Render fornecerá uma URL para o back-end, permitindo o acesso à API por meio dessa URL.

Com isso, o back-end MERN está implantado no servidor Render e pronto para ser acessado na web. Manter a documentação atualizada e realizar manutenções regulares é importante para garantir o correto funcionamento da aplicação. Agora, é possível prosseguir com o desenvolvimento da parte de front-end e integrá-la ao back-end já implantado.

REFERÊNCIAS

EXPRESS DOC. Disponível em https://expressjs.com/pt-br/. Acesso em: 10 out. 2023

LEARN MongoDB in 1 Hour 🍃 (2023). [S.I,]: Bro Code, 2023. Color. Legendado. Disponível em: https://www.youtube.com/watch?v=c2M-rlkkT5o. Acesso em: 16 out. 2023.

MONGODB (O Banco de Dados NoSQL mais Legal) // Dicionário do Programador. Código Fonte Tv, 2021. (9.83 min.), P&B. Disponível em: https://youtu.be/4dTI1mVLX3I?si=_aKGGd1IsQIrnzsI. Acesso em: 16 out. 2023.

MOZILLA Developer. Disponível em: https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introduction. Acesso em: 13 out. 2023

NODE.JS DOC. Disponível em: https://nodejs.org/en/docs. Acesso em: 8 out. 2023

REACT A JavaScript library for Building User Interfaces. Disponível: https://reactjs.org. Acesso em: 18 out. 2023.

REACT.JS The History of React.js on a Timeline. Hámori, Fenerec. Disponível em: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/. Acesso em: 18 out. 2023