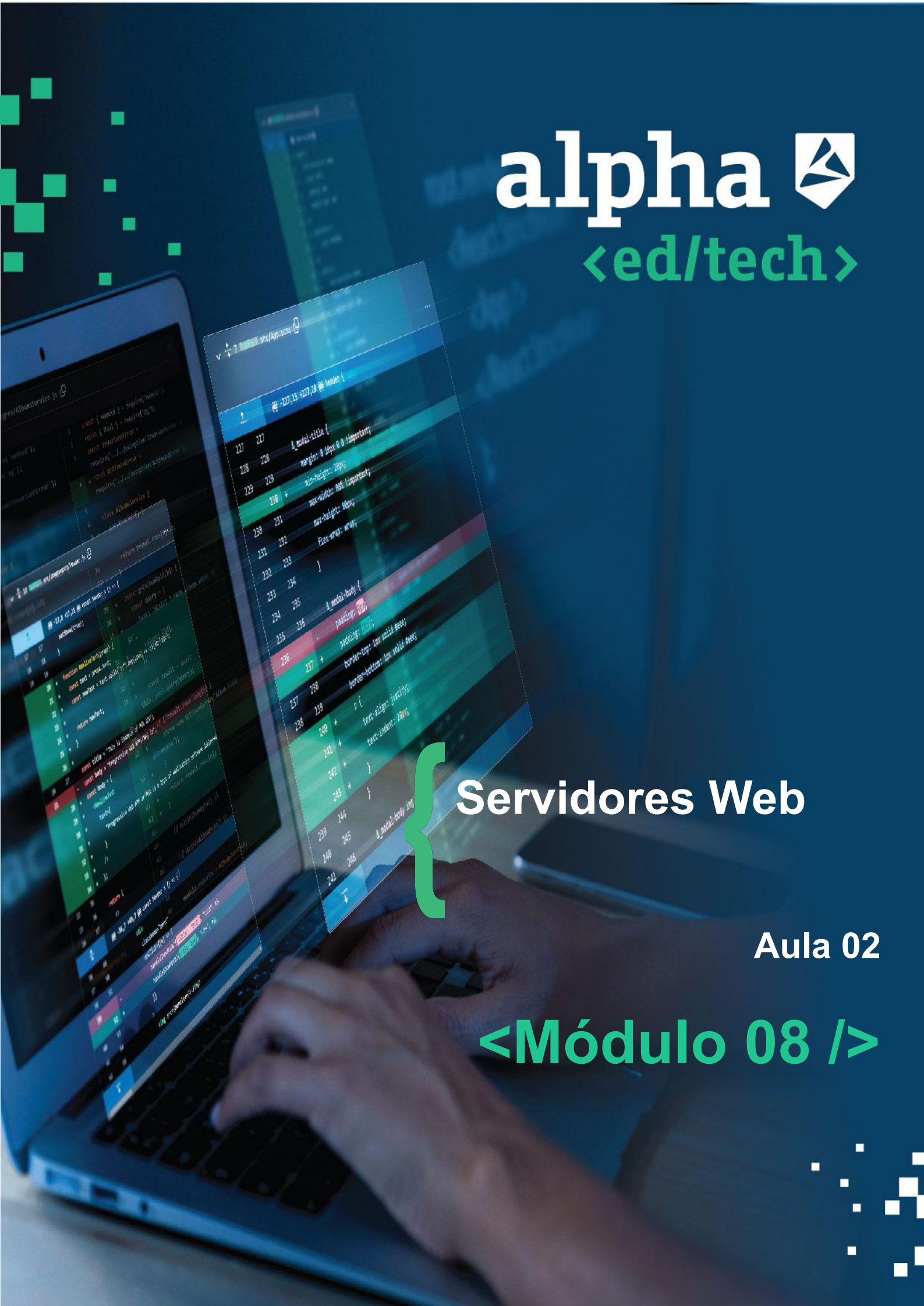

alpha
<ed/tech>
Servidores Web
Aula 02
<Módulo 08 />

Fundamentos de JavaScript para Nodejs

Módulos

Um módulo no Node.js é essencialmente um arquivo JavaScript que encapsula e organiza um conjunto de funcionalidades relacionadas.
Este arquivo pode ser reutilizado em outros scripts, promovendo a modularidade e a estruturação eficiente do código.
Os módulos desempenham um papel crucial na promoção da reutilização de código e na organização lógica de funcionalidades em um projeto Node.js. Ao dividir um aplicativo em módulos, os desenvolvedores podem modularizar o código, facilitando a manutenção, compreensão e escalabilidade do sistema.

CommonJS

No Node.js, o sistema de módulos padrão é baseado no CommonJS, um conjunto de especificações para módulos em JavaScript.
O CommonJS define um formato padrão para organizar e importar módulos em ambientes JavaScript fora do navegador, como no caso do Node.js.
A ideia central do CommonJS é proporcionar um sistema modular para o JavaScript, permitindo a criação de código mais organizado, reutilizável e fácil de manter. Aqui estão alguns conceitos principais do CommonJS:

‘require’ e ‘exports’

Para importar funcionalidades de um módulo para outro, o CommonJS utiliza a função require para importar e exports (ou module.exports) para exportar.
A função require é usada para carregar um módulo existente, enquanto exports é utilizado para expor funcionalidades do módulo atual para outros módulos.

```
// Arquivo chamado 'meuModulo.js'
function minhaFuncao() {
    // lógica da função
}
module.exports = minhaFuncao;
```

```
// Arquivo chamado 'script.js'
const minhaFuncao = require('./meuModulo');
// agora 'minhaFuncao' pode ser utilizada neste script
```

CommonJS x ESM

O Node.js implementa o sistema de módulos CommonJS para permitir a construção de aplicativos escaláveis e modulares. É importante notar que, com o advento do ECMAScript 6 (ES6), também conhecido como ECMAScript 2015, a especificação de módulos do JavaScript foi expandida, introduzindo o import e export. Entretanto, o CommonJS ainda é amplamente utilizado no ecossistema Node.js devido à sua maturidade e compatibilidade.

Fundamentos de JavaScript para Nodejs

Criação de Módulos Personalizados

A criação de módulos personalizados envolve a definição de funções, variáveis e lógica específica em um arquivo separado, que pode ser posteriormente importado em outros scripts.

```
// em um arquivo chamado 'minhaBiblioteca.js'
function funcao1() {
    // lógica da função 1
}

function funcao2() {
    // lógica da função 2
}

module.exports = { funcao1, funcao2 };
```

Módulos Nativos

Em Node.js, os módulos nativos são bibliotecas incorporadas que fornecem funcionalidades essenciais para várias tarefas.
Eles são parte integrante do Node.js e estão disponíveis sem a necessidade de instalação adicional.
A seguir veremos alguns módulos nativos com exemplos de código.

1. Módulo 'fs' (*File System*)

O módulo 'fs' é utilizado para realizar operações de leitura, escrita e manipulação de arquivos no sistema de arquivos.

https://nodejs.org/docs/latest/api/fs.html

```
const fs = require('fs');

// Leitura de um arquivo
fs.readFile('arquivo.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});

// Escrita em um arquivo
fs.writeFile('novoArquivo.txt', 'Conteúdo do novo arquivo', (err) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log('Arquivo criado com sucesso!');
});
```

Fundamentos de JavaScript para Nodejs

2. Módulo ‘http’

O módulo 'http' é usado para criar um servidor HTTP e lidar com requisições e respostas.

https://nodejs.org/docs/latest/api/http.html

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, World!\n');
});

server.listen(3000, '127.0.0.1', () => {
    console.log('Servidor ouvindo em http://127.0.0.1:3000/');
});
```

3. Módulo ‘path’

O módulo 'path' facilita a manipulação e construção de caminhos de arquivos e diretórios.

https://nodejs.org/docs/latest/api/path.html

```
const path = require('path');

const caminhoAbsoluto = path.resolve('pasta', 'arquivo.txt');
console.log(caminhoAbsoluto);
```

4. Módulo ‘events’

O módulo events fornece uma infraestrutura para trabalhar com eventos.

https://nodejs.org/docs/latest/api/events.html

```
const EventEmitter = require('events');

class MeuEmitter extends EventEmitter {}

const meuEmitter = new MeuEmitter();
meuEmitter.on('evento', () => {
  console.log('O evento foi acionado!');
});

meuEmitter.emit('evento');
```

Fundamentos de JavaScript para Nodejs

5. Módulo ‘crypto’

O módulo crypto fornece funcionalidades para criptografia e descriptografia.

https://nodejs.org/docs/latest/api/crypto.html

```
const crypto = require('crypto');

const mensagem = 'Minha mensagem secreta';
const hash = crypto.createHash('sha256').update(mensagem).digest('hex');

console.log('Mensagem:', mensagem);
console.log('Hash SHA-256:', hash);
```

6. Módulo ‘child_process’

O módulo child_process é utilizado para criar processos secundários e interagir com eles.

https://nodejs.org/docs/latest/api/child_process.html

```
const { exec } = require('child_process');

const comando = 'ls';

const processo = exec(comando, (erro, stdout, stderr) => {
  if (erro) {
    console.error(`Erro ao executar o comando: ${erro.message}`);
    return;
  }
  console.log(`Resultado:\n${stdout}`);
  if (stderr) {
    console.error(`Erro no comando:\n${stderr}`);
  }
});

processo.on('exit', (codigo) => {
  console.log(`O comando foi encerrado com o código de saída: ${codigo}`);
});
```

Documentação

Consulte sempre a documentação oficial da versão que estiver utilizando.
Novos módulos são implementados e outros deixam de existir.

https://nodejs.org/docs/latest/api/

Fundamentos de JavaScript para Nodejs

Módulos ‘express’

O Express é um framework para Node.js que simplifica o desenvolvimento de aplicativos web.
Embora o Express seja uma biblioteca externa, ele utiliza algumas bibliotecas nativas do Node.js para realizar suas funcionalidades.
Aqui estão algumas das principais bibliotecas nativas do Node.js que o Express utiliza:

- **http**: O módulo HTTP é uma biblioteca nativa do Node.js que fornece funcionalidades para criar servidores HTTP. O Express utiliza esse módulo para lidar com requisições e respostas HTTP.
- **net**: O módulo net fornece funcionalidades para comunicação de rede. O Express usa isso por trás dos panos para lidar com a comunicação de rede relacionada a servidores.
- **querystring**: Este módulo é usado para manipular dados de consulta em URLs. O Express utiliza isso para analisar e gerar strings de consulta.
- **url**: O módulo URL oferece métodos para trabalhar com URLs. O Express usa isso para analisar URLs e extrair informações relevantes, como caminho e parâmetros de consulta.
- **path**: O módulo Path fornece utilitários para trabalhar com caminhos de arquivo e diretórios. O Express usa isso para manipular e normalizar caminhos de arquivo em suas operações internas.
- **util**: O módulo Util oferece várias funções de utilitário úteis. O Express utiliza isso para várias operações internas, como formatação de mensagens de erro.
- **events**: O módulo Events é fundamental para a construção de sistemas baseados em eventos no Node.js. O Express faz uso disso para criar e manipular eventos em sua arquitetura.
- **stream**: O módulo Stream é usado para lidar com operações de fluxo de dados. O Express usa isso para trabalhar com entrada e saída de dados durante o processamento de solicitações e respostas.

Essas são algumas das bibliotecas nativas do Node.js que o Express utiliza para fornecer um ambiente robusto para o desenvolvimento de aplicativos web.

Express

É importante notar que, embora o Express utilize essas bibliotecas internamente, você geralmente não precisa interagir diretamente com elas ao usar o Express para construir seus aplicativos.
O Express fornece uma camada de abstração que simplifica o desenvolvimento, permitindo que os desenvolvedores se concentrem mais na lógica do aplicativo do que na manipulação de detalhes de baixo nível.

Módulo ‘express’

Criando um servidor web

Apesar de podermos executar diretamente um arquivo Javascript utilizando nodejs, um projeto necessita de uma organização de pastas, arquivos, módulos entre outras boas práticas de estruturação.

Inicialização do Projeto

Abra o terminal e crie uma nova pasta para o seu projeto.
Navegue até a pasta no terminal e execute o seguinte comando para iniciar um novo projeto Node.js:

```
npm init -y
```

Este comando criará automaticamente um arquivo package.json com configurações padrão.

Instalação do Express

Agora, você precisa instalar o Express. No terminal, execute o seguinte comando:

```
npm install express --save
```

Isso instalará o Express e adicionará a dependência ao seu arquivo package.json.

Criação do Arquivo Principal (app.js)

Crie um arquivo chamado app.js na raiz do seu projeto. Este será o arquivo principal do seu aplicativo Express.

```
1   // app.js
2   const express = require('express');
3   const app = express();
4   const porta = 3000;
5
6   app.get('/', (req, res) => {
7     res.send('Olá, Express!');
8   });
9
10  app.listen(porta, () => {
11    console.log(`Servidor está rodando em http://localhost:${porta}`);
12  });
```

Módulo ‘express’

Adição de um Script no ‘package.json’

Abra o seu arquivo package.json e adicione um script de execução. Substitua o conteúdo atual pelo seguinte:

```
{
    "name": "seu-projeto",
    "version": "1.0.0",
    "description": "Descrição do seu projeto",
    "main": "app.js",
    "scripts": {
      "start": "node app.js"
    },
    "keywords": [],
    "author": "Seu Nome",
    "license": "ISC",
    "dependencies": {
      "express": "^4.17.1"
    }
}
```

Execução do Aplicativo

Agora, você pode iniciar seu aplicativo executando o seguinte comando no terminal:

```
npm start
```

Isso iniciará o servidor Express e você verá a mensagem "Servidor está rodando em http://localhost:3000". Abra um navegador e vá para http://localhost:3000 para ver o resultado.

Middleware em Nodejs

Introdução

Middleware em Node.js refere-se a um componente fundamental na arquitetura de uma aplicação, responsável por lidar com requisições HTTP, processar dados e facilitar a comunicação entre diferentes partes do sistema. Essa camada intermediária desempenha um papel crucial na construção de aplicações robustas e escaláveis.

Funções do Middleware

O middleware age como um filtro ou manipulador entre o pedido do cliente (request) e a resposta do servidor (response).
Ele é inserido no fluxo da requisição, permitindo a execução de lógicas específicas antes ou depois da resposta ser enviada de volta ao cliente.

Características

- **Modularidade:** Middleware em Node.js é modular, o que significa que você pode criar pequenos módulos independentes para lidar com tarefas específicas. Cada middleware pode focar em uma funcionalidade particular, promovendo a reutilização de código e a fácil manutenção do sistema.

- **Encadeamento:** É possível encadear vários middlewares em uma rota ou em todo o aplicativo, permitindo a execução sequencial de diferentes lógicas. Isso é valioso para dividir responsabilidades e manter o código organizado.

Exemplos

- **Logger Middleware:** Um middleware de registro pode ser utilizado para registrar informações sobre cada requisição, como método, URL, e timestamp. Isso ajuda na depuração e monitoramento da aplicação.

```
const loggerMiddleware = (req, res, next) => {
    console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
    next();
};
```

- **Autenticação Middleware:** Middleware de autenticação pode verificar se um usuário está autenticado antes de permitir o acesso a determinadas rotas.

```
const authenticationMiddleware = (req, res, next) => {
    if (req.isAuthenticated()) {
        return next();
    }
    res.status(401).send('Unauthorized');
};
```

Middleware em Nodejs

Exemplo de uso do middleware no ‘express’

Para aplicar um middleware em uma rota específica ou em todo o aplicativo, você utiliza o método use do Express, um framework popular para construção de aplicações web em Node.js.

```
const express = require('express');
const app = express();

// Middleware de autenticação
const authenticationMiddleware = (req, res, next) => {
  // Verifica se o usuário está autenticado
  const isAuthenticated = true; // Aqui, você substituiria por lógica real de autenticação

  if (isAuthenticated) {
    // Se autenticado, continua para a próxima middleware ou rota
    next();
  } else {
    // Se não autenticado, retorna uma resposta de não autorizado
    res.status(401).send('Acesso não autorizado');
  }
};

// Middleware de log para todas as requisições
const logMiddleware = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
};

// Aplicando o middleware de log para todas as requisições
app.use(logMiddleware);

// Rota pública
app.get('/', (req, res) => {
  res.send('Bem-vindo à página inicial!');
});

// Rota protegida por middleware de autenticação
app.get('/restrito', authenticationMiddleware, (req, res) => {
  res.send('Você acessou a rota restrita!');
});

// Rota para lidar com outros casos
app.get('/outra-rota', (req, res) => {
  res.send('Outra rota pública!');
});

// Iniciando o servidor
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
});
```

‘express Router’ em Nodejs

Introdução

O Router é uma maneira de organizar as rotas em uma aplicação Express, permitindo dividir a lógica de roteamento em módulos independentes e reutilizáveis. Isso é especialmente útil para aplicações maiores e mais complexas.

Criando um arquivo roteador

```js
// routes.js
const express = require('express');
const router = express.Router();

// Rota principal
router.get('/', (req, res) => {
  res.send('Bem-vindo à rota principal!');
});

// Rota de exemplo
router.get('/exemplo', (req, res) => {
  res.send('Esta é uma rota de exemplo.');
});

module.exports = router;
```

Usando o arquivo roteador criado

```js
// app.js (ou seu arquivo principal)
const express = require('express');
const app = express();

// Importe o roteador
const routes = require('./routes');

// Use o roteador para as rotas que começam com '/api'
app.use('/api', routes);

// Inicie o servidor
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Servidor rodando na porta ${PORT}`);
});
```

No exemplo acima, as requisições feitas para a rota ‘/api/’ e ‘/api/exemplo’ serão tratadas pelo sistema de roteamento do express.

API RESTful em Nodejs

Introdução

Node.js é essencial na criação de APIs RESTful devido à sua arquitetura assíncrona, permitindo a execução eficiente de operações de I/O não bloqueantes e garantindo tempos de resposta rápidos.
Sua natureza baseada em eventos possibilita a construção de APIs altamente escaláveis, capazes de lidar com várias conexões simultâneas de maneira eficaz. A flexibilidade do JavaScript em todo o stack simplifica a colaboração entre equipes de desenvolvimento.
Além disso, o vasto ecossistema de módulos via npm agiliza o desenvolvimento, enquanto a comunidade ativa oferece suporte contínuo. Em resumo, o Node.js é uma escolha crucial para quem busca criar APIs RESTful eficientes, escaláveis e robustas.

Vantagens da escolha por nodejs

- **Desempenho Elevado:** O Node.js é construído sobre o motor V8 do Google Chrome, conhecido por sua eficiência e desempenho. Ele utiliza uma arquitetura de I/O não bloqueante que permite lidar com várias operações simultaneamente, resultando em tempos de resposta rápidos.

- **Natureza Assíncrona:** O Node.js é baseado em uma arquitetura assíncrona, o que significa que operações de entrada e saída (I/O) não bloqueiam a execução do código. Isso é especialmente benéfico para lidar com operações intensivas em I/O, como acesso a banco de dados, leitura de arquivos e chamadas de API, sem prejudicar o desempenho.

- **JavaScript no Lado do Servidor:** Utilizar JavaScript tanto no lado do cliente quanto no lado do servidor simplifica o desenvolvimento, pois os desenvolvedores podem usar a mesma linguagem em todo o stack. Isso também facilita a colaboração entre equipes de desenvolvimento front-end e back-end.

- **Ecossistema NPM Abundante:** O Node.js possui um gerenciador de pacotes robusto chamado npm, que oferece acesso a uma ampla variedade de bibliotecas e módulos. A vasta comunidade de desenvolvedores contribui regularmente com novos pacotes, facilitando a integração de funcionalidades adicionais nas APIs.

- **Facilidade de Aprendizado:** Para desenvolvedores que já estão familiarizados com JavaScript, a transição para o Node.js é relativamente suave. Isso reduz a curva de aprendizado para aqueles que já trabalham com tecnologias web.

- **Escalabilidade Horizontal Eficiente:** O Node.js é conhecido por sua capacidade de escalar horizontalmente, facilitando o gerenciamento de um grande número de conexões simultâneas. Isso é particularmente útil em casos de aplicações que exigem escalabilidade para lidar com tráfego intenso.

- **Comunidade Ativa e Suporte Contínuo:** O Node.js possui uma comunidade grande e ativa. Isso significa que há uma quantidade significativa de recursos, tutoriais, fóruns e suporte disponíveis online. A comunidade contribui para a evolução contínua da tecnologia.

- **Desenvolvimento de APIs RESTful Simples:** O Node.js é frequentemente utilizado para criar APIs RESTful, seguindo os princípios de arquitetura web. Isso resulta em APIs que são fáceis de entender, escaláveis e interoperáveis, permitindo uma comunicação eficiente entre sistemas distribuídos.

- **Flexibilidade de Plataformas de Hospedagem:** Node.js é compatível com diversas plataformas de hospedagem, como AWS, Heroku, e Microsoft Azure. Essa flexibilidade permite escolher o ambiente de execução mais adequado às necessidades do projeto.

- **Manutenção Simplificada:** A consistência do uso de JavaScript em todo o stack facilita a manutenção do código. Além disso, o modelo de módulos do Node.js promove a modularidade e reutilização de código, simplificando a manutenção e a evolução do sistema.

API RESTful em Nodejs

RESTful na comunicação web

Representational State Transfer (RESTful) é um paradigma arquitetural que se tornou fundamental para o design de serviços web eficientes e escaláveis.
Desenvolvido por Roy Fielding em sua tese de doutorado, REST se baseia em princípios simples, mas poderosos, para facilitar a comunicação entre sistemas distribuídos.

Em sua essência, REST utiliza os métodos padrão do protocolo HTTP, como GET, POST, PUT e DELETE, para realizar operações em recursos identificáveis por URLs.

A abordagem centrada em recursos permite uma representação clara do estado da aplicação, com a comunicação sendo representada de maneira uniforme e sem estado. Isso promove a escalabilidade e a interoperabilidade entre diferentes sistemas, tornando REST uma escolha prevalente para o desenvolvimento de APIs.

As características chave do REST incluem a arquitetura cliente-servidor, onde o cliente e o servidor operam independentemente, a statelessness, que implica que cada requisição do cliente contém todas as informações necessárias para ser compreendida pelo servidor, e a visão dos recursos como entidades identificáveis com URIs, acessíveis por meio de operações HTTP padrão.

Métodos HTTP

Os métodos HTTP (Hypertext Transfer Protocol) são a base da comunicação entre clientes e servidores na web.
Cada método representa uma operação específica que o cliente solicita ao servidor em relação a um recurso identificado por uma URL.

Método GET

Descrição:
- O método GET é utilizado para solicitar dados de um recurso específico.

Características:
- É seguro e idempotente, o que significa que múltiplas requisições GET para o mesmo recurso produzirão o mesmo resultado.
- As informações são enviadas no cabeçalho da URL, sendo visíveis na barra de endereços do navegador.
- Não deve ser usado para operações que causem alterações no estado do servidor.

Método POST

Descrição:
- O método POST é utilizado para submeter dados para serem processados a um recurso específico.

Características:
- Não é idempotente, ou seja, várias requisições POST podem ter resultados diferentes.
- Os dados são enviados no corpo da requisição, permitindo o envio de uma quantidade maior de informações.
- Usado para criar novos recursos no servidor ou realizar operações que causem alterações no estado.

API RESTful

Ao adotar princípios RESTful, os desenvolvedores podem criar APIs web robustas, flexíveis e de fácil manutenção, proporcionando uma base sólida para a comunicação eficaz entre sistemas distribuídos na era da web moderna.

API RESTful em Nodejs

Método PUT

Descrição:
- O método PUT é utilizado para atualizar um recurso específico ou criar um recurso se ele não existir.

Características:
- É idempotente, ou seja, várias requisições PUT para o mesmo recurso produzirão o mesmo resultado.
- Os dados são enviados no corpo da requisição, similar ao POST.
- Geralmente usado para atualizar completamente um recurso ou criar um novo recurso se o identificador (URL) não existir.

Método DELETE

Descrição:
- O método DELETE é utilizado para solicitar a remoção de um recurso específico.

Características:
- É idempotente.
- Geralmente usado para excluir um recurso identificado pela URL.
- Como o nome sugere, implica na remoção do recurso no servidor.

SOAP

SOAP (*Simple Object Access Protocol*) é um protocolo de comunicação usado em serviços web para troca de informações entre sistemas.
Embora SOAP tenha sido amplamente utilizado no passado, há algumas razões pelas quais algumas pessoas optam por não usá-lo em serviços web modernos. Algumas dessas razões incluem:

- **Complexidade:** SOAP pode ser considerado mais complexo do que alternativas mais leves, como REST *(Representational State Transfer*). A estrutura de mensagem SOAP é mais pesada e pode incluir muitos elementos desnecessários para certos cenários.
- **Overhead:** SOAP tende a ter um overhead maior devido à sua estrutura XML robusta e à necessidade de processamento adicional. Isso pode resultar em mensagens maiores e, consequentemente, em um consumo maior de largura de banda.
- **Verbosidade:** SOAP é conhecido por ser mais verbose (verborrágico), devido à sua natureza baseada em XML. Isso significa que as mensagens podem ser mais longas e difíceis de ler em comparação com formatos de mensagem mais simples, como JSON no caso de serviços REST.
- **Menos suporte nativo em linguagens modernas:** Muitas linguagens de programação modernas têm melhor suporte integrado para formatos de mensagem mais leves, como JSON. Isso facilita a serialização e desserialização de dados em serviços web.
- **Flexibilidade:** Serviços web RESTful são frequentemente considerados mais flexíveis, uma vez que podem utilizar diferentes formatos de mensagem (JSON, XML, etc.) e métodos HTTP (GET, POST, PUT, DELETE) de forma mais direta.
- **Padrões de Indústria:** Em muitos casos, padrões de mercado e indústria favorecem REST sobre SOAP. Muitas APIs públicas e serviços web modernos optam por REST devido à sua simplicidade e fácil integração.

API RESTful em Nodejs

Exemplo

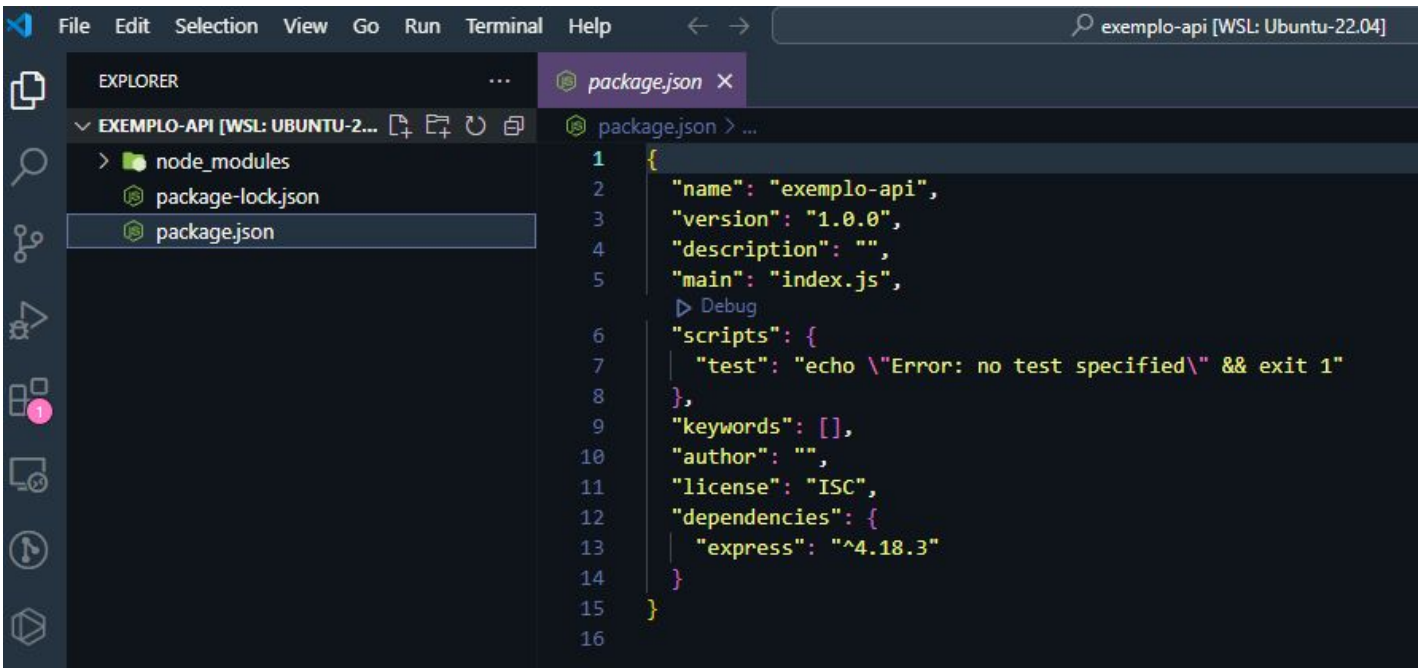
Neste exemplo desejamos criar uma API RESTful em nodejs que utilize a biblioteca 'express' para a entidade 'users' que possui 3 (três) propriedades: 'id', 'username', 'email'.
A ideia é que, por meio desta API o usuário possa realizar as operações de CRUD ('Create', 'Read', 'Update' e 'Delete') dos usuários cadastrados.
Não utilizaremos banco de dados mas os usuários serão armazenados em um array de objetos.

Criando o projeto

```
kenji@DESKTOP-0EELV37:~$ pwd
/home/kenji
kenji@DESKTOP-0EELV37:~$ mkdir exemplo-api
kenji@DESKTOP-0EELV37:~$ cd exemplo-api
kenji@DESKTOP-0EELV37:~/exemplo-api$ npm init -y
Wrote to /home/kenji/exemplo-api/package.json:

{
  "name": "exemplo-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


kenji@DESKTOP-0EELV37:~/exemplo-api$ npm install express --save

added 64 packages, and audited 65 packages in 3s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
kenji@DESKTOP-0EELV37:~/exemplo-api$ code .
```

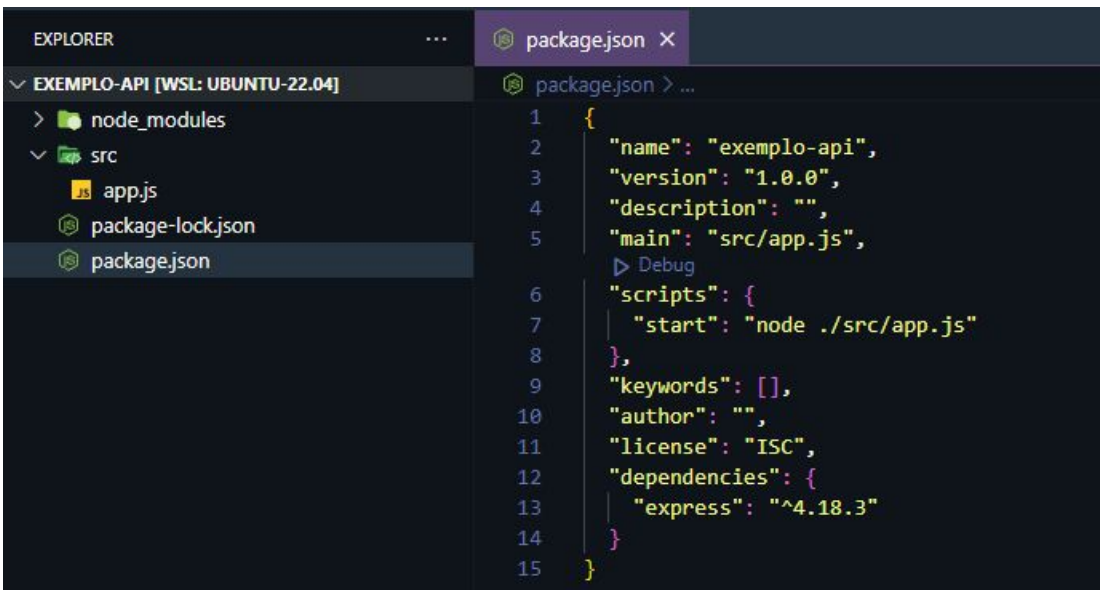
Inicialmente verificou-se a pasta do usuário.
Da pasta atual, foi criada a pasta de nosso projeto, neste caso com o nome 'exemplo-api'
Com o comando 'npm init –y' foi iniciado o projeto com as configurações padrão.
Como iremos utilizar o express, instalamos a dependência com o comando 'npm install express –save'
Iniciamos o VSCode a partir da pasta do projeto.

API RESTful em Nodejs



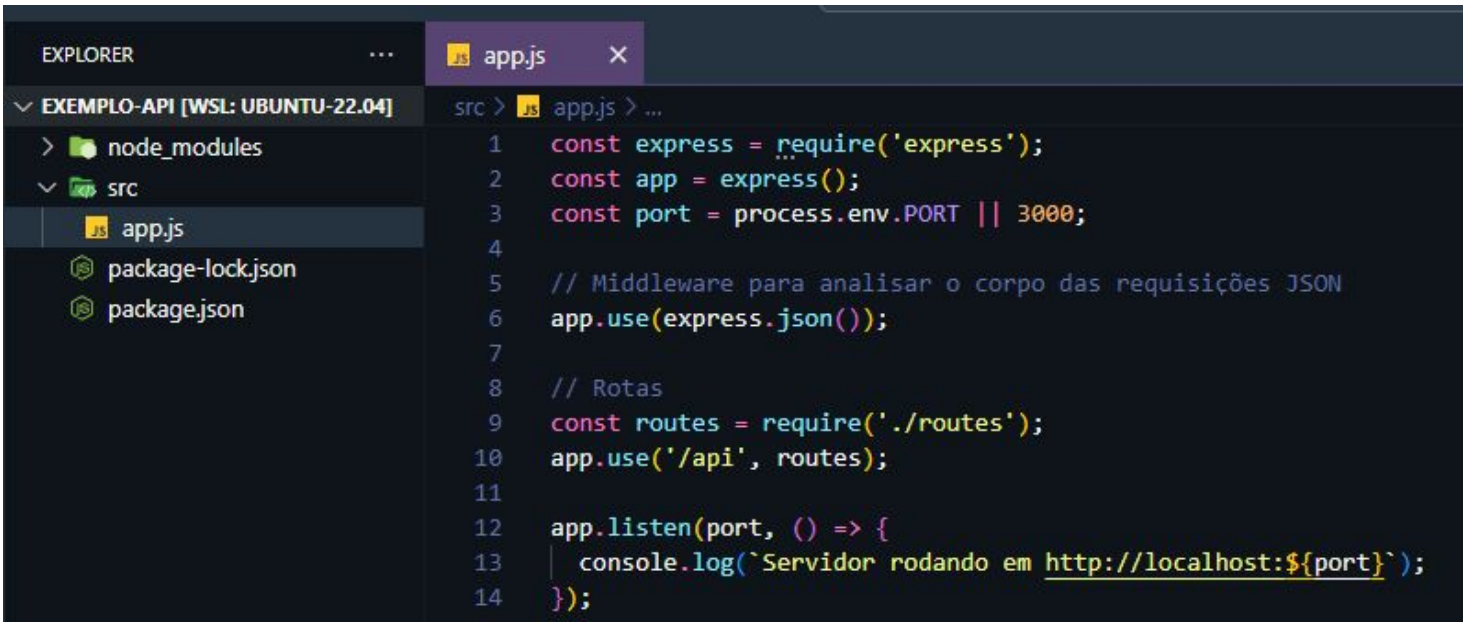
Note que a pasta 'exemplo-api' já está selecionada.
O arquivo 'package.json' que já discrimina a dependência do 'express' na versão '4.18.3' que foi a instalada neste exemplo.
Você poderá ajustar os dados do seu projeto alterando os valores dos atributos do JSON.
Também note que a pasta 'node_modules' contém as dependências de projeto. Esta pasta usualmente contém muitos arquivos e, para fins de controle de versão git, ela é ignorada pois não há a necessidade de ficar armazenando as dependências pois o que importa é o código da aplicação.

Ajustando o arquivo package.json



Neste caso, alteramos o atributo "main", apagamos o script "test" e adicionamos o script "start" que será responsável para executar nossa API.
Também criamos a pasta "src" e dentro dela criamos o arquivo "app.js"

API RESTful em Nodejs



O arquivo 'app.js' agora contém o carregamento da biblioteca 'express' por meio de carregamento de dependências do tipo commonJS na constante 'express'.

A constante 'app' irá possuir a instância da nossa API.

A constante 'port' define a porta '300' se não tiver a variável 'ENV' definida no ambiente do processo.
- Se quiser mudar a porta de execução da API na inicialização do seu 'app.js' bastaria substituir o script de 'start' para algo como 'PORT=5000 node ./src/app.js'.
- Neste caso 'process.env.PORT possuirá o valor '5000' e a constante 'port' conterá '5000' e não '3000'.

Na linha 6 veja que utilizamos um middleware que irá tratar as requisições que contiverem no corpo ('body') dados no formato JSON.
- Quando você utiliza app.use(express.json()) no Express.js, você está configurando o middleware express.json().
- Este middleware é responsável por analisar o corpo das solicitações HTTP que contêm dados no formato JSON.
- Ao definir esse middleware, o Express.js automaticamente analisa o corpo das solicitações recebidas, interpretando o conteúdo como JSON e transformando-o em um objeto JavaScript.
- Isso é particularmente útil ao lidar com solicitações POST ou PUT, onde os dados enviados pelo cliente estão no formato JSON, como é comum em APIs RESTful.

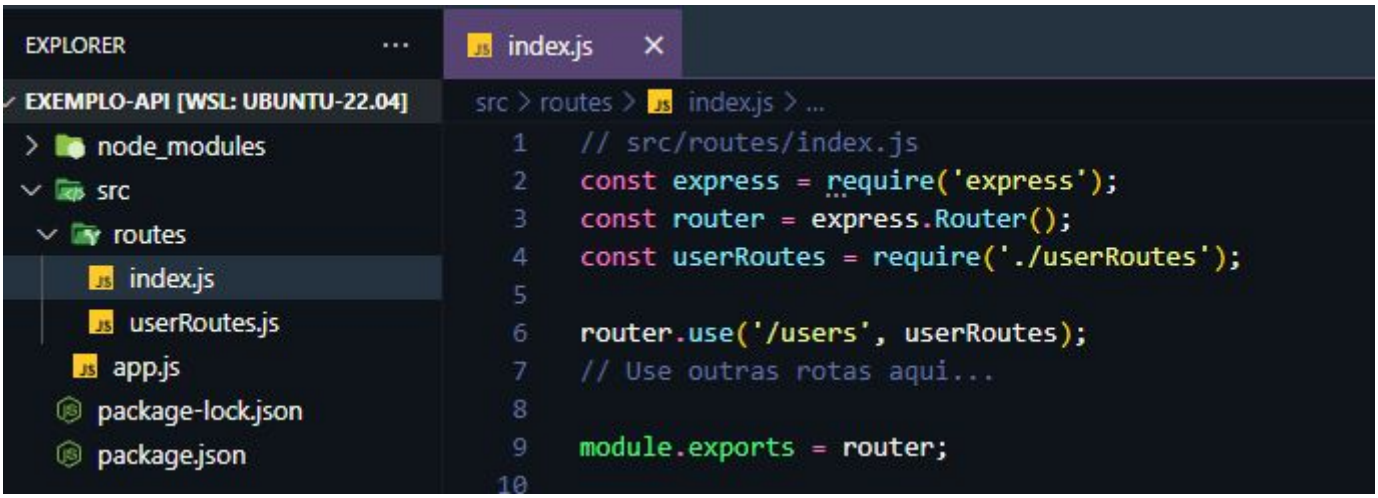
Na linha 9 carregamos as rotas que estarão definidas na pasta 'routes' que ainda não foi criada.

Na linha 10 utilizamos outro middleware que interceptará a rota '/api' e direcionará para o arquivo de gerenciamento de rotas 'index.js' que criaremos a seguir.

express

É de extrema importância que você entenda cada passo da construção de uma aplicação com 'express', principalmente a estrutura de gestão de rotas e disposição de pastas.
Os projetos irão utilizar outras estruturas de pastas conforme a necessidade individual pois um projeto bem organizado facilitará a identificação e manutenção de código por outros desenvolvedores.

API RESTful em Nodejs



Criamos agora a pasta 'routes' e dentro dela dois arquivos: 'index.js' e 'userRoutes.js'.

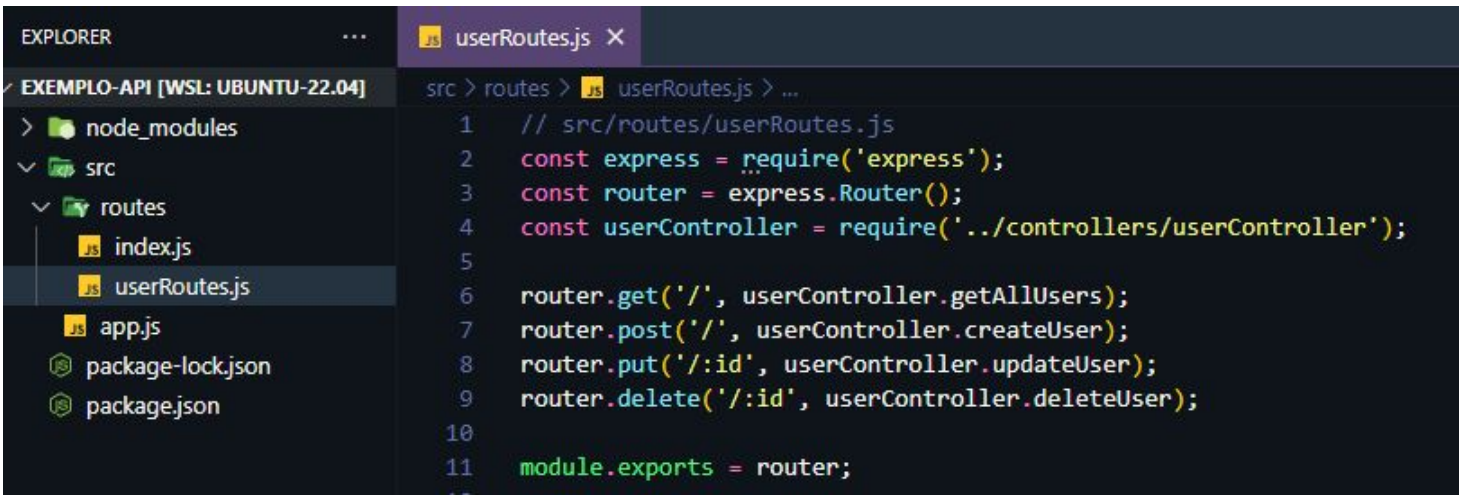
- Note que o uso de 'index.js' é um padrão para que quando você realiza a importação da pasta '/routes' ele automaticamente procura por este arquivo não precisando escrever o caminho completo, como por exemplo: const routes = require("./routes/indes.js")

No arquivo 'index.js' veja que carregamos na linha 3 o gerenciador de rotas que é exportada na linha 9.

Na linha 4 a constante userRoutes recebe uma função que gerenciará as rotas específicas para tratamento de usuários.

Na linha 6 adicionamos a função de callback 'userRoutes' no middleware de forma a interceptar a rota '/api/users' e que tratará as requisições.

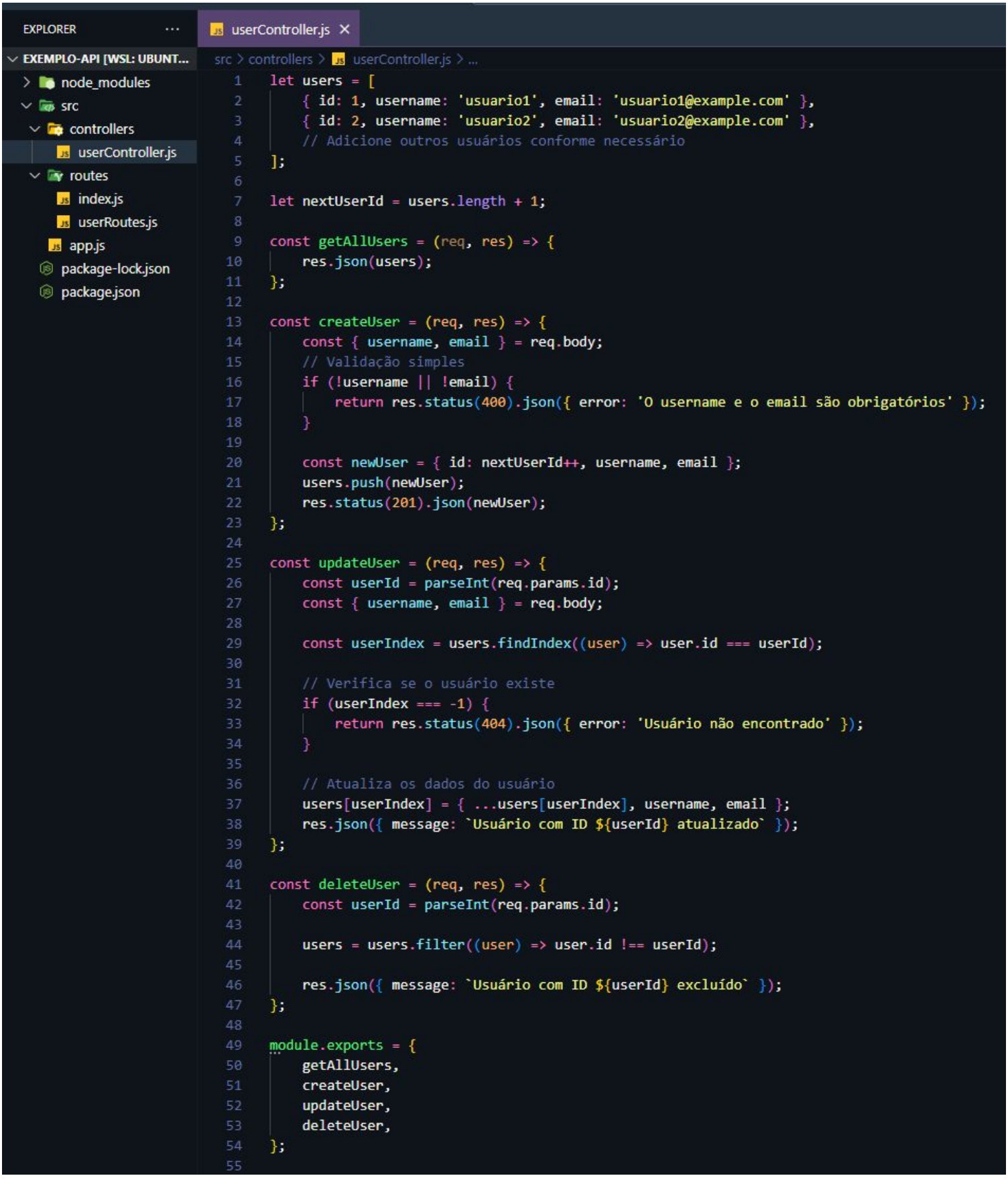
- Neste exemplo, apesar de ser voltado apenas para tratamento da entidade de usuários, poderíamos estender para outras entidades, isto é, adicionar outras rotas;
- Também é importante notar que apesar deste middleware descrever '/users', este middleware foi chamado de '/api'. Assim, a rota final ficaria '/api/users'.



O arquivo 'userRoutes.js' que contém o tratamento das requisições do tipo GET, POST, PUT, DELETE conforme padrão RESTful, intercepta as requisições a esta rota '/' e direciona para a execução dos 'controllers', neste caso criaremos uma pasta 'controllers' e dentro dela os controladores específicos de usuário 'userController.js'.

API RESTful em Nodejs

userController.js



API RESTful em Nodejs

userController.js

Iniciamos criando uma variável que poderá sofrer alterações de nome 'users' que contém um array de objetos.

- Foram criados dois usuários com atributos 'id', 'username' e 'email';
- Veja que este array poderia ser os dados contidos em um banco de dados como faremos em módulos posteriores de manipulação de banco de dados.

A variável 'nextUserId' inicia com o valor do tamanho do array 'users' + 1.

- Como está declarado neste local, a contagem sempre será incrementada independente do tamanho do array em caso de exclusões de usuários.

'getAllUsers' que retorna os usuários no formato JSON;

'createUser' que cria um usuário;

- Neste caso 'criar' um usuário é representado pela adição de um objeto ao array 'users' por meio de push;
- Esta função valida a existência dos valores de 'username' e 'email' e caso não exista um deles retorna erro '400' e um JSON com a explicação do erro a quem fez a requisição.
- Caso tenha sucesso, cria um objeto 'newUser' que contém um 'id' único adicionando-se '1' à variável 'nextUserId'

'updateUser' que faz a atualização do usuário por meio do parâmetro 'id' recebido

- Note que o 'id' é recebido como parâmetro no arquivo 'userRoutes.js', neste caso bastaria passar o id logo após o '/', isto é, '/api/users/1' com o método PUT faria a atualização do usuário de 'id' igual a '1';
- Veja também que o uso do middleware no arquivo 'app.js', 'express.json()' fez com que os valores enviados no corpo da requisição fossem obtidos no 'req.body';
- Para entender a linha 27 é necessário lembrar que ela poderia ser reescrita em duas linhas:
  const username = req.body.username;
  const email = req.body.email;
- Faz-se o uso de users.findIndex para localizar o índice que representa a igualdade de um elemento de 'users' no qual o 'id' é o mesmo passado pelo parâmetro da linha 26.
- Caso não se encontre, o índice conterá '-1' o que resulta em um retorno de erro de usuário não encontrado;
- No caso de sucesso, na linha 37 há a substituição do elemento de índice encontrado em 'userIndex' com os dados recebidos pelo usuário;

'deleteUser' apaga o usuário

- Recebe-se o parâmetro id e armazena em 'userId';
- Faz-se o filtro excluindo-se o usuário que possui o 'id' enviado e retorna a mensagem de usuário excluído com sucesso;

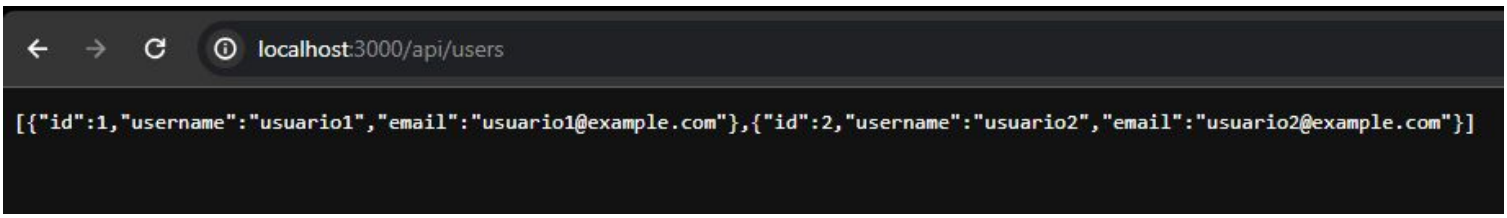
API RESTful em Nodejs

Executando o projeto

Para executar o projeto, basta ir na pasta do projeto e digitar 'npm start'

```
kenji@DESKTOP-0EELV37:~/exemplo-api$ npm start

> exemplo-api@1.0.0 start
> node ./src/app.js

Servidor rodando em http://localhost:3000
```

Para requisições do tipo GET pode-se realizar uma chamada direta pelo navegador:



Para as demais requisições (POST, PUT, DELETE) há a necessidade de utilização de requisições por meio de programação tradicional de acesso a API por meio de AJAX.