**NODE**

**NPM – Node Package Manager**

**Criar npm**

npm init -y

**Installar pacotes:**

npm install {nome\_do\_pacote}

npm install express

**Atualizar pacotes:**

npm update

**Alterar a dependencies de um package (dev, prod):**

npm install express --save-dev // muda para devDependencies

npm install express --save-prod // muda para dependencies

**Instalar versão específica de package:**

npm install express@2.1.0 -E // -E para remover o ‘^’

**Desinstalar um pacakge:**

npm uninstall express

**Listar packages:**

npm ls // mostra os packages e suas dependencias

npm ls -depth=0 // mostra somente os packages instalados

npm outdated // mostra os packages desatualizados

**MANIPULANDO ARQUIVOS – FS**

**Leitura de pastas e arquivos:**

Importando o módulo:

const *fs* = require('fs').*promises*; // importa o módulo fs como promises

Criamos uma função para ler os arquivos e pastas de um caminho:

async function readdir(*rootDir*) {

*rootDir* = *rootDir* || path.resolve(\_\_dirname);

    const files = await *fs*.readdir(*rootDir*);

    walk(files, *rootDir*);

}

**\*IMPORTANTE:** fs.readdir retorna os arquivos e pastas do caminho informado

Na função walk, criamos uma lógica com fs.stat (que retorna status do arquivo lido) para podermos criar uma leitura recursiva dos itens.

async function walk(*files*, *rootDir*) {

    for (let file of *files*) {

// cria um caminho completo do arquivo

        const fileFullPath = path.resolve(*rootDir*, file);

        const stats = await *fs*.stat(fileFullPath);

        if (/\.git/g.test(fileFullPath)) continue; // re

        if (/node\_modules/g.test(fileFullPath)) continue; // re

// se o arquivo for uma pasta, ele recomeça a leitura

        if (stats.isDirectory()) {

            readdir(fileFullPath);

            continue;

        }

        if (!/\.html$/g.test(fileFullPath)) continue; // re

        console.log(fileFullPath);

    }

}

**Leitura e escrita de arquivos:**

***Escrita:***

Por meio de modulos, criamos um arquivo para a escrita de novos arquivos, que exporta um função que recebe o caminho onde o arquivo será escrito e os dados que irão conter nele:

const *fs* = require('fs').*promises*;

*module*.*exports* = (*caminho*, *dados*) => {

*fs*.writeFile(*caminho*, *dados*, { flag: 'w' });

}

**\*IMPORTANTE**: a flag é opcional (por padrão ‘w’)

No arquivo princial (app.js), criamos o caminho do arquivo a ser escrito e um array com objetos para servir de conteúdo. Nesse caso foi criado um arquivo .json, então utilizamos o método JSON.stringify() para converter o array de objetos para a formatação JSON:

const path = require('path');

const caminhoArquivo = path.resolve(\_\_dirname, 'teste.json');

const escreve = require('./modules/escreve');

const pessoas = [

    { nome: 'João' },

    { nome: 'Maria' },

    { nome: 'Eduardo' },

    { nome: 'Luiza' },

];

const json = JSON.stringify(pessoas, '', 2);

escreve(caminhoArquivo, json);

***Leitura:***

Para a leitura, foi criado um arquivo que exporta uma função com o fs.readFile, que serve para ler o conteúdo de um arquivo:

const *fs* = require('fs').*promises*;

*module*.*exports* = (*caminho*) => *fs*.readFile(*caminho*, 'utf-8');

\*IMPORTANTE: o fs.readFile() recebe o caminho do arquivo que será lido e a codificação que será utilizada (padrão: utf8)

Novamente no arquivo principal (app.js), importamos a função de leitura do modulo ler e criamos uma async function, pois a função de leitura retorna uma promise. Dentro dessa async function, enviamos o resultado da promise para uma outra função que irá transformar os dados JSON em formato JS utilizando o método JSON.parse(), onde podemos fazer o que quisermos com os dados lidos (no exemplo eles foram apenas logados na tela):

const path = require('path');

const caminhoArquivo = path.resolve(\_\_dirname, 'teste.json');

const ler = require('./modules/ler');

async function lerArquivo(*caminho*) {

    const arqLido = await ler(*caminho*);

    renderizaDados(arqLido);

}

function renderizaDados(*dados*) {

*dados* = JSON.parse(*dados*);

*dados*.forEach(*val* => console.log(*val*));

}

lerArquivo(caminhoArquivo);

**EXPRESS:**

O express é um módulo que é utilizado para gerenciar de forma prática as rotas do servidor de um site. Ele precisa ser instalado no node utilizando o comando npm i express.

Por padrão, para importar o empress utilizamos as seguintes nomenclaturas:

const express = require('express');

const app = express();

Para entender melhor o express, é importante conhecer o CRUD, que representa as operações que podem ser feitas no servidor:

CREATE, READ, UPDATE, DELETE -> significado literal de CRUD

POST    GET   PUT     DELETE -> elementos que representam CRUD no express

GET -> é uma solicitação de leitura

POST -> solicitação criação/envio de alguma informação

PUT -> solicitação de atualização de informação

DELETE -> solitação para deletar informação

Com o express, podemos definir o que cada solicitação pode fazer nas rotas que criamos para o site. Exemplo:

app.get('/', (*req*, *res*) => {

*res*.send(`

    <form action="/" method="POST">

    Nome: <input type="text" name="nome">

    <button>Enviar</button>

    `);

});

\* req e res significam, respectivamente, requisição e resposta do servidor

Aqui temos a definição de como a solitação GET irá se comportar na rota ‘/’, que seria a home do site. Aqui, utilizando o res.send(), estamos enviando como resposta um HTML que contém um formulário.

Para que esse formulário possa ser enviado, precisamos utilizar o método POST, como no exemplo abaixo:

app.post('/', (*req*, *res*) => {

*res*.send('Recebi o formulario!');

})

\*nesse caso o form irá carregar esse POST pois é para onde o action do form está apontando

Aqui configuramos o POST da rota ‘/’, que será exebido ao enviarmos com sucesso o formulário.

Segue um outro exemplo de configuração de rota, agora da aba contato do site:

app.get('/contato', (*req*, *res*) => {

*res*.send('Obrigado por entrar em contato com a gente!');

});

Para que o servidor fique ativo e possamos visualizar o site, fazemos com que o express utilize uma porta do servidor com o método **app.listen()** . Em area de teste, contuma-se utilizar portas que não são muito usadas por aplicações normais, como 3000 ou 3333:

app.listen(3000, () => {

    console.log('Acessar http://localhost:3000');

    console.log('Servidor executando na porta 3000');

});

**NODEMON:**

O nodemon é um package do node que serve para auxiliar no uso do express. Com ele, não é necessário ficar reiniciando o servidor toda vez que uma alteração for feita, já que ele faz essa atualização automaticamente.

Para instalar ele é só utilizar a seguinte linha de código no terminal:

npm i nodemon --save-dev

Para ativar o nodemon, vamos alterar o start do no nosso package.json:

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "nodemon server.js"

  },

\*caso não haja o start, adicionamos ele, caso já exista, só trocar node para nodemon

**req.params, req.query**

Nas urls, além das rotas, podemos ter também outras informações. Duas delas seriam os parametros e os queries strings, que podem ser identificados da seguinte forma:

//        rota         rota     param  query             query

// http://facebook.com/profiles/12345?campanha=googleads&nome\_camp=seila

Para permitir que um parametro ou query string seja adicionado no url, adicionamos eles da seguinte forma:

app.get('/testes/:idUsuarios?/:parametros?', (*req*, *res*) => {

    console.log(*req*.query);

*res*.send(*req*.query);

})

Utilizamos o ‘:’ no começo da identificador e o ‘?’ no final para indicar que o link pode receber um param ou query nesse local.

**req.body**

O req.body é utilizado no método POST, quando vamos receber o valor de um formulário enviado por exemplo. Ex:

app.post('/', (*req*, *res*) => {

    console.log(*req*.body);

*res*.send(`O que você me enviou foi: ${*req*.body.nome}`);

});

No exemplo acima, o req.body teoricamente recebe o POST que foi disparado, e logamos ele na tela. Porém, para que o req.body receba o valor disparado no POST, é preciso configurar o app.use, da seguinte forma:

app.use(express.urlencoded({ extended: true }));

Com isso, o req.body recebe o valor do POST, como foi configurado.

**Routes e Controllers**

Se aproximando mais de uma aplicação de node real, temos que para uma organização e controle melhor das rotas utilizamos Routes e Controllers.

Para iniciar a utilização, criamos um arquivo routes.js no nosso projeto e adicionamos as seguintes importações:

const express = require('express');

const route = express.Router();

Esse Router() será responsável por controlar as rotas do nosso servidor:

Primeiramente, criamos um controller para cada rota do nosso site, que é onde irá conter as funções de req e res:

Texto

Descrição gerada automaticamente

**homeController.js**:

*exports*.paginaInicial = (*req*, *res*) => {

*res*.send(`

        <form action="/" method="POST">

        Nome: <input type="text" name="nome">

        <button>Enviar</button>

        `);

};

*exports*.trataPost = (*req*, *res*) => {

*res*.send('Sou sua nova rota de POST');

};

**contatoController.js**:

*exports*.paginaInicial = (*req*, *res*) => {

*res*.send(`Obrigado por entrar em contato!`);

};

Após isso, importamos os controllers dentro de routes e criamos as rotas do site, mas ao invés de declarar como app.get ou app.post, declaramos como route.get / route.post:

const express = require('express');

const route = express.Router();

const homeController = require('./controllers/homeController')

const contatoController = require('./controllers/contatoController')

// Rotas da home

route.get('/', homeController.paginaInicial);

route.post('/', homeController.trataPost);

// Rotas de contato

route.get('/contato', contatoController.paginaInicial);

*module*.*exports* = route;

\*exportamos o route no final do código, sobrescrevendo o module.exports;

E para finalizar, importamos o file routes.js no nosso arquivo principal e no local dos app.get e app.post que havíamos feitos, vamos inserir somente a linha de código app.use(routes):

const express = require('express');

const app = express();

const routes = require('./routes'); // arquivo importado

app.use(express.urlencoded({ extended: true }));

app.use(routes); // linha adicionada

app.listen(3000, () => {

    console.log('Acessar http://localhost:3000');

    console.log('Servidor executando na porta 3000');

});

**Criando views:**

Para adicionar as views do nosso site primeiramente vamos adicionar no script principal a seguinte configuração:

app.set('views', path.resolve(\_\_dirname, 'src', 'views'));

app.set('view engine', 'ejs');

\*ejs é o formato escolhido para o arquivo template html

Instalar ejs ou formato escolhido no npm:

npm install ejs

Após isso, criamos uma pasta src para adicionar os controllers e as views. EM views criamos um arquivo index.ejs (a extensão escolhida no app.set(‘view engine’):

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

**Criando pasta para arquivo estático:**

Para os arquivos estáticos vamos criar uma pasta com o nome *public* e adicionar a seguinte configuração no script principal:

app.use(express.static(path.resolve(\_\_dirname, 'public')));

**Middlewares:**

Os middlewares são funções/ações que podem ser executadas antes da resposta final ao cliente. Ele pode executar qualquer ação e pode tanto ser destinado a uma requisição específica (como somente no GET de uma rota) ou pode estar presente de forma global em todas as requisições (mais comum).

Segue um exemplo de middleware:

*exports*.middlewareGlobal = (*req*, *res*, *next*) => {

    if (*req*.body.cliente) {

*req*.body.cliente = *req*.body.cliente.replace('Miranda', 'NÃO USE MIRANDA');

        console.log();

        console.log(`Vi que você postou ${*req*.body.cliente}`);

        console.log();

    }

    next();

};

Nesse middleware global, verificamos se o cliente realizou um POST (req) do campo cliente, interceptando a mensagem e alterando o seu valor caso contenha o nome ‘Miranda’ no POST enviado.

Neste exemplo, podemos entender o poder e importância que um middleware pode ter, podendo interceptar envios de cliente ou realizar ações específicas de acordo com o que o cliente faça ou envie.

Para importar esse middleware no script principal utilizamos o app.use, como no exemplo:

const { middlewareGlobal } = require('./src/middlewares/middleware');

app.use(middlewareGlobal);

Com isso, o middleware é aplicado em todos as rotas e suas requisições.

**Conexão com MongoDB Cloud:**

link: mongodb+srv://guimatds:<db\_password>@cursojs1.ulfbq.mongodb.net/?retryWrites=true&w=majority&appName=cursojs1

**MongoDB – Conexão e primeiro model**

Para conectar o nosso projeto com o banco de dados do MongoDB vamos ter que realizar algumas operações. A primeira delas vai ser instalar o Mongoose, utilizando o npm i mongoose.

Após isso, vamos importar o mongoose no nosso script principal, logo abaixo da importação do app:

const express = require('express');

const app = express();

const *mongoose* = require('mongoose');

Na hora de realizar a conexão, vamos precisar do link de conexão do MongoDB, que podemos colocar em uma constante.

const connectionString = 'mongodb+srv://guimatds:<senha>@cursojs1.ulfbq.mongodb.net/BASEDEDADOS?retryWrites=true&w=majority&appName=cursojs1';

**\*IMPORTANTE:** alterar o campo <senha> para a senha do usuário do BD

Por último, utilizamos o método connect para nos conectar com o banco de dados:

*mongoose*.connect(connectionString)

    .then(() => {

        console.log('BD conectado')

        app.emit('pronto');

    })

    .catch(*e* => console.log(*e*));

Como o mongoose.connect nos retorna uma promise, podemos utilizar um then para fazer com a conexão com o BD ocorra antes do servidor iniciar. Para isso usamos o app.emmit() enviando um sinal, que será recebido pelo app.on() e assim o que tiver dentro dele será executado, que no nosso caso, será o listen do app:

app.on('pronto', () => {

    app.listen(3000, () => {

        console.log('Acessar http://localhost:3000');

        console.log('Servidor executando na porta 3000');

    });

});

**EJS:**

O EJS é um formato de arquivo que permite que você crie lógicas de JS dentro de um código HTML, além de algumas outras funcionalidades ([documentação](https://ejs.co)).

Antes de falar sobre o EJS, vale falar que na resposta das solitações do nosso site (nos controllers), podemos enviar um objeto com dados que podem ser utilizados pelo EJS, como no exemplo:

*exports*.paginaInicial = (*req*, *res*) => {

*res*.render('index', {

        titulo: 'Este será o título da página',

        numeros: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    });

    return

};

Nesse exemplo, estamos enviando a string titulo e o array números, que podem ser utilizados dentro dos arquivos .ejs no projeto.

É possível também enviarmos também uma variável para o ejs para todas as rotas do servidor ao invés de enviar somente para uma específica, realizando o envio por um middleware global:

*exports*.middlewareGlobal = (*req*, *res*, *next*) => {

*res*.locals.umaVariavelLocal = 'Este é o valor da variável local';

    next();

};

Um exemplo de acesso:

<section *class*="container">

        <h1>

            <%- typeof titulo !== 'undefined' ? titulo : undefined %>

        </h1>

        <form *action*="/" *method*="post">

            <label>Cliente</label>

            <input *type*="text" *name*="cliente">

            <button *type*="submit">Submit</button>

        </form>

    </section>

Note que para utilizarmos as funcionalidades do ejs precisamos utilizar tags um pouco diferentes da que utilizamos no HTML. Segue alguns exemplos de tags:

<% : Utilizada para códigos no geral. Não exibe nenhum resultado (sem outout)

<%= : Exibe o valor no template (com HTML escapado)

<%- : Exibe o valor no template, mas com o HTML aplicado

**Includes:**

Dentro do EJS, podemos criar templates com códigos que vão se repitir em várias páginas e dar includes nesses templates, evitando assim a repetição de código.

O processo padrão é criar dentro de views uma pasta com o nome includes e adicionar dentro dela esses arquivos padrões:

Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente

Um exemplo de um include:

<!DOCTYPE *html*>

<html *lang*="pt-br">

<head>

    <meta *charset*="UTF-8">

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1.0">

    <title>Um teste</title>

</head>

<body>

Aqui temos a parte do head de um arquivo HTML, que se repete em todo código HTML (com algumas alterações) e que podemos criar um include para evitar a repetição.

No arquivo que vamos adicionar o include, realizamos da seguinte forma:

<%- include('includes/head') %>

    <section *class*="container">

        <h1>

            <%- typeof titulo !== 'undefined' ? titulo : undefined %>

        </h1>

...

Utilizando a tag <%- chamamos a função include e passamos o nome do arquivo que queremos incluir nesse local.