Estas anotações são sobre o bootcamp Desenvolvedor Front-End da instituição IGTI iniciado em 20/11/2020.

<https://online.igti.com.br/courses>

Sumário

[MÓDULO 01 3](#_Toc58407675)

[Aula 01 - Introdução 4](#_Toc58407676)

[Aula 02 - Node.js 5](#_Toc58407677)

[Aula 03 – Noções HTML 7](#_Toc58407678)

[Aula 04 – Noções CSS 8](#_Toc58407679)

[Aula 05 - Introdução ao JavaScript 9](#_Toc58407680)

[Aula 06 - JavaScript - comandos de bloco 10](#_Toc58407681)

[Aula 07 - JavaScript - manipulação do DOM 13](#_Toc58407682)

[Aula 08 - JavaScript - formulários e manipulação de eventos 15](#_Toc58407683)

[Aula 09 - CRUD com HTML, CSS e JavaScript 18](#_Toc58407684)

[Aula 10 - JavaScript moderno – Introdução 27](#_Toc58407685)

[Aula 11 - JavaScript moderno - Manipulação de arrays 29](#_Toc58407686)

[Aula 12 - JavaScript moderno - Rest/Spread operator e destructuring 32](#_Toc58407687)

[Aula 13 - Refatoração do projeto de CRUD 34](#_Toc58407688)

[Aula 14 - Introdução à programação assíncrona com JavaScript 35](#_Toc58407689)

[Aula 15 - Funções setTimeout e setInterval 36](#_Toc58407690)

[Aula 16 - Requisições HTTP com JavaScript 37](#_Toc58407691)

[MÓDULO 02 39](#_Toc58407692)

[Aula 01 – Apresentação do curso 40](#_Toc58407693)

[Aula 02 – Conceitos e termos 41](#_Toc58407694)

[Aula 03 - Instalação do Node.js - Visual Studio Code 42](#_Toc58407695)

[Aula 04 - Primeira App React 43](#_Toc58407696)

[Aula 05 - Conhecendo nossa App 44](#_Toc58407697)

[Aula 06 - Calculadora 46](#_Toc58407698)

[Aula 07 - Estilizando a nossa calculadora 50](#_Toc58407699)

[Aula 08 - Dando funções à nossa Calculadora 56](#_Toc58407700)

[Aula 09 - Axios 65](#_Toc58407701)

# MÓDULO 01

## Aula 01 - Introdução

## Aula 02 - Node.js

Para iniciar um projeto node.js é necessário criar um pacote do tipo .json, para isto, basta abrir o cmd, navegar até a pasta em que se deseja criar os pacotes e então digitar o seguinte:

npm init -y

E estará criado o projeto node.js em formato .json.

Existem dois tipos de dependências:

* Dependency: o tipo de dependência padrão local;
* Dev-dependency: o tipo de dependência que é utilizado pelo desenvolvedor apenas para o desenvolvimento, ou seja, isso não será utilizado no deploy quando a aplicação for para o servidor de fato;

Para criar uma dev-dependency para desenvolver com node.js, basta navegar pelo terminal até a pasta desejada e digitar:

npm install live-server –-save-dev

Então este comando irá inserir no package.json uma área responsável pelo controle de dependências de desenvolvimento (devDependencies); e também irá criar um arquivo, que não deve ser modificado, chamado package-lock.json que é responsável pelo controle de versões destas dependências.

Isto é de extrema utilidade pois cria um arquivo .json que grava quais dependências o projeto tem e caso seja necessário trocar de máquina o projeto, movendo apenas estes arquivos e executando o comando ‘npm install’ o npm instalará todas as dependências necessárias.

Para rodar o live-server, primeiro deve-se configurar que ele será responsável por rodar a aplicação e para isto, deve-se adicionar a key “start” no package.json com o valor “live-server”, ficando assim:

{

  "name": "aula-02-node-js",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

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

    "start": "live-server"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "devDependencies": {

    "live-server": "^1.2.1"

  }

}

E então no terminal, na pasta em que o arquivo se encontra rodar:

npm start

Pronto, o navegador abrirá a aplicação.

## Aula 03 – Noções HTML

## Aula 04 – Noções CSS

## Aula 05 - Introdução ao JavaScript

## Aula 06 - JavaScript - comandos de bloco

Sintaxes:

// Comparando 2 números com if/else

var a = 3;

var b = 5;

if (a < b) {

  console.log(a + ' é menor que ' + b);

} else {

  if (a > b) {

    console.log(a + ' é maior que ' + b);

  } else {

    console.log(a + ' e ' + b + ' são iguais');

  }

}

// Validação de dia com if/else

var day = 1;

if (day === 1) {

  console.log('Domingo');

} else {

  if (day === 2) {

    console.log('Segunda-feira');

  } else {

    if (day === 3) {

      console.log('Terça-feira');

    } else {

      if (day === 4) {

        console.log('Quarta-feira');

      } else {

        if (day === 5) {

          console.log('Quinta-feira');

        } else {

          if (day === 6) {

            console.log('Sexta-feira');

          } else {

            if (day === 7) {

              console.log('Sábado');

            } else {

              console.log('Dia inválido');

            }

          }

        }

      }

    }

  }

}

// Validação de dia com switch

switch (day) {

  case 1:

    console.log('Domingo');

    break;

  case 2:

    console.log('Segunda-feira');

    break;

  case 3:

    console.log('Terça-feira');

    break;

  case 4:

    console.log('Quarta-feira');

    break;

  case 5:

    console.log('Quinta-feira');

    break;

  case 6:

    console.log('Sexta-feira');

    break;

  case 7:

    console.log('Sábado');

    break;

  default:

    console.log('Dia inválido');

}

// Comparando 2 números com operador ternário

var compareResult = a > b ? 1 : a < b ? -1 : 0;

// Verificando dia da semana com operador ternário

var weekDay =

  day === 1

    ? 'Domingo'

    : day === 2

    ? 'Segunda-feira'

    : day === 3

    ? 'Terça-feira'

    : day === 4

    ? 'Quarta-feira'

    : day === 5

    ? 'Quinta-feira'

    : day === 6

    ? 'Sexta-feira'

    : day === 7

    ? 'Sábado'

    : 'Dia inválido';

## Aula 07 - JavaScript - manipulação do DOM

**document.querySelector()**

Para manipular elementos da página HTML via JS utiliza-se o método do objeto document ‘querySelector’, ele é responsável por selecionar um ou vários elementos da página HTML, estes podendo ser de três tipos:

var input1 = document.querySelector('#idGenerico');

input1.value = 'Raphael Gomide';

var input2 = document.querySelector('.classeGenerica');

input3.value = 'Abilonson';

var input3 = document.querySelector('elementoHTML'); //<p>, <h1>, etc...

input3.value = 'Jurema';

**document.querySelectorAll();**

Semelhante ao ‘querySelector()’ porém retorna uma nodeList com todos os elementor de um tipo especificado no argumento do método;

const important = document.querySelectorAll('urgent');

Uma maneira interessante de se manipular o conteúdo do texto de um elemento é após capturar o elemento desejado usando um querySelector, utiliza-se o método ‘.textContent();’ para alterar seu valor.

const important = document.querySelectorAll('urgent');

important.textContent = 'Olá Mundo! sou importante!';

Aqui é interessante mostrar que é possível manipular o CSS através do JS. Isto é feito através das classes em CSS.

Supões que se deseja alterar a formatação de um texto quando ele for clicado, para isto utilizamos a propriedade onclick no HTML que será responsável por executar esta função, e então criamos uma classe CSS com a propriedade desejada para então escrever uma função para adicionar essa classe à respectiva tag HTML.

Ficando com:

HTML:

<h1 *id*="toBeEmphasized" *onclick*="clicked(event)" *class*="">Olá Mundo!</h1>

CSS:

*.emphasis* {

  font-weight: bold;

  color: darkgreen;

}

JS:

function clicked() {

  const text = document.querySelector('#toBeEmphasized');

  classList = *Array*.from(text.classList);

  //Se tiver a classe emphasis a função irá remover esta classe, caso contrário irá adicionar:

  if (classList.includes('emphasis')){

    text.classList.remove('emphasis');

    text.textContent = 'Sem o ".empahsis" nas classes deste objeto';

  } else {

    text.classList.add('emphasis');

    text.textContent = '".emphasis" foi adicionado';

  }

}

Este foi um exemplo de manipulação simples de CSS de um elemento HTML utilizando JS, demonstrando o método ‘remove()’ e ‘add()’ da classList do elemento HTML.

## Aula 08 - JavaScript - formulários e manipulação de eventos

Para criar formulários em HTML utiliza-se a tag <form> e nele podem ser inseridos os campos que serão utilizados.

<!DOCTYPE *html*>

<html *lang*="en">

  <head>

    <meta *charset*="UTF-8" />

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

    <title>Formulário</title>

  </head>

  <body>

    <form>

      <p>

        <textarea *rows*="10" *cols*="10">Conteúdo</textarea>

      </p>

      <p>

        <input *type*="text" *id*="input1" *placeholder*="Informe o seu nome" />

      </p>

      <p>

        <input *type*="email" *id*="input2" *placeholder*="Informe o seu email" />

      </p>

      <p>

        <input *type*="date" *id*="input3" *placeholder*="Informe o seu email" />

      </p>

      <p>

        <input *type*="checkbox" *id*="input4" *placeholder*="Informe o seu nome" />

        Rock

      </p>

      <p>

        <input

*type*="radio"

*name*="opcao"

*id*="input2"

*placeholder*="Informe o seu nome"

        />

        Sim

        <input

*type*="radio"

*id*="input2"

*name*="opcao"

*placeholder*="Informe o seu nome"

        />

        Não

      </p>

      <input *type*="submit" *value*="Enviar" />

    </form>

    <script *src*="./js/script.js"></script>

  </body>

</html>

Para manipular estes elementos com o JS basta fazer da mesma maneira como se manipularia qualquer outro elemento HTML, através do querySelector:

var input1 = document.querySelector('#input1');

input1.value = Bleblio;

Então o valor do imput1 será setado para ‘Bleblio’.

A função adEventListener() também desempenha um papel muito importante, é através dela que ao se executar uma ação na página o script em JS é executado.

function clicked() {

  const text = document.querySelector('#toBeEmphasized');

  classList = *Array*.from(text.classList);

  //Se tiver a classe emphasis a função irá remover esta classe, caso contrário irá adicionar:

  if (classList.includes('emphasis')){

    text.classList.remove('emphasis');

    text.textContent = 'Sem o ".empahsis" nas classes deste objeto';

  } else {

    text.classList.add('emphasis');

    text.textContent = '".emphasis" foi adicionado';

  }

};

function start() {

  let textInput = document.querySelector('#textInput');

  let form = document.querySelector('form');

  textInput.addEventListener('keyup', countTextLength);

  form.addEventListener('submit', preventSubmit);

}

function countTextLength(*event*) {

  let count = *event*.target.value.length;

  let span = document.querySelector('#textLength');

  span.textContent = count;

}

function preventSubmit(*event*) {

*event*.preventDefault();

  alert('ainda não estamos funcionando, volte mais tarde!')

}

start();

## Aula 09 - CRUD com HTML, CSS e JavaScript

Parte 01:

Nesta aula se aprende a manipular a inserção de elementos HTML através do JS. Isto é útil pois as vezes se deseja ciar uma lista ou coisa similar que sejam anexados elementos quando alguma acao é tomada; no exemplo quando um nome é digitado e apertado ENTER este é anexado a uma lista HTML.

A primeira coisa a se fazer é evitar que a página seja recarregada automaticamente ao dar o submit do form, para isto criamos uma função que é executada ao executarmos o submit do form que previne esta ação.

function start(){

  preventFormSubmit();

};

function preventFormSubmit(){

  function handleFormSubmit(*event*){

*event*.preventDefault();

  }

  const form = document.querySelector('form');

  form.addEventListener('submit', handleFormSubmit);

}

start();

Nesta lógica, assim que o script for executado ele executará a função start() que será responsável por executar a preventFormSubmit();

Então com o adEventListener observando o form, sempre que for dado um ‘submit’ nele ele executará o handleFormSubmit, que recebe o ‘submit’ como argumento e roda o ‘submit’.preventDefault();

Parte 02:

Agora devemos pegar os dados inseridos no formulário, para anexa-los a uma lista de nome. Para isso é interessante ter uma função responsável por observar o formulário e assim que a tecla enter for pressionada pegar estes dados e salvar em um array para depois apresentá-los na tela.

Para ficar responsável por pegar estes dados criamos a função activateInput(), que observa e faz o push no array e o render() que renderiza esse array na tela

//variaveis globais

let inputName;

let globalNames = ['test'];

//funçoes gerais

function preventFormSubmit() {

  //prevent refresh da página

  function handleFormSubmit(*event*) {

*event*.preventDefault();

  }

  const form = document.querySelector('form');

  form.addEventListener('submit', handleFormSubmit);

}

function activateInput() {

  //observa a tecla enter e quando pressionada adiciona ao globalNames o que for digitado no formulário

  function handleTyping(*event*) {

    if (*event*.key === 'Enter') {

      globalNames.push(*event*.target.value);

      render();

    }

  }

  inputName.focus();

  inputName.addEventListener('keyup', handleTyping);

}

function render() {

  //renderiza as informácoes do aray globalNames em forma de lista, através de push e appendChild

  let divNames = document.querySelector('#names');

  divNames.innerHTML = '';

  let ul = document.createElement('ul');

  for (let i = 0; i < globalNames.length; i++) {

    let currentName = globalNames[i];

    let li = document.createElement('li');

    li.textContent = currentName;

    ul.appendChild(li);

  }

  divNames.appendChild(ul);

}

//script a ser executado quando a página for carregada

function start() {

  //apelidos

  inputName = document.querySelector('#inputName');

  //açoes a serem executadas

  preventFormSubmit();

  activateInput();

  render();

}

start();

O principal aqui é como utiliza-se o createElement() e appendChild para criar e anexar os elementos HTML.

Parte 03:

Inicia-se fazendo alguma otimizações como limpar a tela ao renderizar uma nova lista.

Feito isto a deve-se criar a função de remoção de elemento a partir do botão, para isso utilizaremos o mesmo principio de createElement para a lista só que para o botão e para facilitar o entendimento do código, cria-se uma função para criar tal botão. Lembrando que após a remoção do item é necessário renderizar a lista novamente.

//variaveis globais

let inputName;

let globalNames = ['test'];

//funçoes gerais

function preventFormSubmit() {

  //prevent refresh da página

  function handleFormSubmit(*event*) {

*event*.preventDefault();

  }

  const form = document.querySelector('form');

  form.addEventListener('submit', handleFormSubmit);

}

function activateInput() {

  //observa a tecla enter e quando pressionada adiciona ao globalNames o que for digitado no formulário

  function handleTyping(*event*) {

    if (*event*.key === 'Enter') {

      globalNames.push(*event*.target.value);

      render();

    }

    // if (event.key === 'ArrowUp') {

    //  console.log(globalNames);

    // }

  }

  inputName.focus();

  inputName.addEventListener('keyup', handleTyping);

}

function render() {

  function createDeleteButton(*index*) {

    function deleteName() {

      // console.log(globalNames[index])

      globalNames.splice(*index*, 1);

      render();

    }

    const button = document.createElement('button');

    button.textContent = 'x';

    button.classList.add('deleteButton');

    button.addEventListener('click', deleteName);

    return button;

  }

  //renderiza as informacoes do aray globalNames em forma de lista, através de push e appendChild

  let divNames = document.querySelector('#names');

  divNames.innerHTML = '';

  let ul = document.createElement('ul');

  for (let i = 0; i < globalNames.length; i++) {

    let currentName = globalNames[i];

    let li = document.createElement('li');

    let button = createDeleteButton(i);

    let span = document.createElement('span');

    span.textContent = currentName;

    li.appendChild(button);

    li.appendChild(span);

    ul.appendChild(li);

  }

  divNames.appendChild(ul);

  clearInput();

}

function clearInput() {

  inputName.value = '';

  inputName.focus();

}

//script a ser executado quando a página for carregada

function start() {

  //apelidos

  inputName = document.querySelector('#inputName');

  //açoes a serem executadas

  preventFormSubmit();

  activateInput();

  render();

}

start();

Parte 04:

Agora iremos mexer com a edição dos itens da lista. Para isto criamos uma função responsável por passar o valor do item para a barra de input e uma variável global para verificar se o item está sendo editado.

//variaveis globais

let inputName;

let currentIndex = null;

let globalNames = ['test','foo','spam'];

let isEditing = false;

//funçoes gerais

function preventFormSubmit() {

  //prevent refresh da página

  function handleFormSubmit(*event*) {

*event*.preventDefault();

  }

  const form = document.querySelector('form');

  form.addEventListener('submit', handleFormSubmit);

}

function activateInput() {

  //observa a tecla enter e quando pressionada adiciona ao globalNames o que for digitado no formulário

  function updateName(*newName*){

    //responsável por fazer o update do item selecionado

    console.log(newName);

    console.log(currentIndex);

    globalNames[currentIndex] = *newName*;

  }

  function handleTyping(*event*) {

    //verifica se existe texto no input e deixa dar submit para a lista

    let hasText = !!*event*.target.value && *event*.target.value !== '';

    if(!hasText){

      clearInput();

      return;

    }

    if (*event*.key === 'Enter') {

      if(isEditing) {

        updateName(*event*.target.value);

      } else {

        globalNames.push(*event*.target.value);

      }

      render();

      isEditing = false;

      clearInput();

    }

  }

  inputName.focus();

  inputName.addEventListener('keyup', handleTyping);

}

function render() {

  //renderiza as informacoes do aray globalNames em forma de lista na tela, através de push e appendChild

  function createDeleteButton(*index*) {

    function deleteName() {

      // ppermite deletar o item da lista

      globalNames.splice(*index*, 1);

      render();

    }

    const button = document.createElement('button');

    button.textContent = 'x';

    button.classList.add('deleteButton');

    button.addEventListener('click', deleteName);

    return button;

  }

  function createSpan(*name*, *index*) {

    //cria o span onde o texto é armazenado

    function editItem() {

      //permite edicao do texto

      inputName.value = *name*;

      inputName.focus();

      isEditing = true;

      currentIndex = *index*;

    }

    let span = document.createElement('span');

    span.classList.add('clickable');

    span.textContent = *name*;

    span.addEventListener('click', editItem);

    return span;

  }

  let divNames = document.querySelector('#names');

  divNames.innerHTML = '';

  let ul = document.createElement('ul');

  for (let i = 0; i < globalNames.length; i++) {

    let currentName = globalNames[i];

    let li = document.createElement('li');

    let button = createDeleteButton(i);

    let span = createSpan(currentName, i)

    li.appendChild(button);

    li.appendChild(span);

    ul.appendChild(li);

  }

  divNames.appendChild(ul);

  clearInput();

}

function clearInput() {

  //limpa o input box

  inputName.value = '';

  inputName.focus();

}

//script a ser executado quando a página for carregada

function start() {

  //apelidos

  inputName = document.querySelector('#inputName');

  //açoes a serem executadas

  preventFormSubmit();

  activateInput();

  render();

}

start();

## Aula 10 - JavaScript moderno – Introdução

A partir do ECMAScript6 o JS introduz alguns conceitos novos:

* Declaração de variáveis: var, const e let;
* Arrow functions;
* Template literals;
* Default parameters;

O código a seguir demonstra estas novas características:

'use strict';

//vas x let x const

//var tem escopo abrangente;

function withVar() {

  for (var i = 0; i < 10; i++) {

    console.log('var - ' + i);

  }

  i = 'sou o "i" criado dentro do loop for';

  console.log('sou o consolog fora do loop for: ' + i + '. withVar()');

}

//let tem escopo reduzido;

function withLet() {

  for (let i = 0; i < 10; i++) {

    console.log('let: ' + i);

  }

  i = 'sou o "i" criado dentro do loop for';

  console.log('sou o consolog fora do loop for: ' + i + '. withLet()');

}

//const

const c = 10;

console.log(c);

c = 5; //Uncaught TypeError: Assignment to constant variable.

console.log(c);

//execução

withVar();

withLet(); //Uncaught ReferenceError: i is not defined

E aqui o arrow function, default parameters e template literals:

//arrow function

const sumArrow1 = (*a*, *b*) => {return sum(*a*,*b*);};

//ou quando só tem um return

const sumArrow2 = (*a*, *b*) =>  *a* + *b*;

//template literals

const nome = 'Leo';

const texto = `Meu nome é ${nome}`;

console.log(texto); //Meu nome é Leo

//default parameters

const

## Aula 11 - JavaScript moderno - Manipulação de arrays

Aqui são demonstrados os principal métodos de manipulação de arrays que o JS oferece:

* map : (Gera um novo array, transformando dados);
* filter : (Gera um novo array filtrando elementos conforme condição);
* forEach (percorre todos os elementos do array aplicando uma lógica/função);
* reduce : (realiza cálculo iterativo com base nos elementos);
* find : (encontra o primeiro elemento de acordo com critério pré-definido);
* some : (retorna booleano se há pelo menos um elemento que atenda à preposição);
* every : (retorna booleano se todos os elementos atendem à preposição);
* sort : (Ordena os elementos com base em um critério);

//função que é executada ao iniciar a página

function start() {

  console.log('.map(): ',doMap());

  console.log('.filter(): ',doFilter());

  console.log('.forEach(): ',doForEach());

  console.log('.reduce(): ',doReduce());

  console.log('.find(): ',doFind());

  console.log('.some(): ',doSome());

  console.log('.every(): ',doEvery());

  console.log('.sort(): ',doSort());

}

//execução da página

start();

//Demonstração dos métodos de arrays mais importantes:

function doMap() {

  const nameEmailArray = people.results.map((*person*) => {

    return {

      name: `${*person*.name.first} ${*person*.name.last}`,

      email: *person*.email,

    };

  });

  return nameEmailArray;

}

function doFilter() {

  const olderThan50 = people.results.filter((*person*) => {

    return *person*.dob.age >= 50;

  });

  return olderThan50;

}

function doForEach() {

  const mappedPeople = doMap();

  let res = [];

  const nameSize = mappedPeople.forEach((*person*) => {

    res.push({ name: *person*.name, nameLength: *person*.name.length });

    return res;

  });

  return res;

}

function doReduce() {

  const totalAges = people.results.reduce((*accumulator*, *current*) => {

    return *accumulator* + *current*.dob.age;

  }, 0);

  return totalAges;

}

function doFind() {

  const mo = people.results.find((*person*) => {

    return *person*.location.state == 'Minas Gerais';

  });

  return mo;

}

function doSome() {

  const mo = people.results.some((*person*) => {

    return *person*.location.state == 'Amazonas';

  });

  return mo;

}

function doEvery() {

  const every = people.results.every((*person*) => {

    return *person*.nat == 'BR';

  });

  return every;

}

function doSort() {

  const mappedPeople = people.results.map((*person*) => {

    return { name: *person*.name.first };

  });

  const filteredPeople = mappedPeople.filter((*person*) =>

*person*.name.startsWith('A')

  );

  const sortedPeople = filteredPeople.sort((*a*, *b*) => {

    return *a*.name.length - *b*.name.length;

  });

  return sortedPeople;

}

## Aula 12 - JavaScript moderno - Rest/Spread operator e destructuring

O operador ‘...’ é responsável pela manipulação de elementos dentro de um array.

* O spread pega um array e ‘espalha’ ele dentro de um array ou algum outro lugar;
* O rest serve para pegar infinitos argumentos em um único array;
* O destructuring é um jeito amis fácil de se pegar vária propriedades de uma variável em apenas uma linha;

function start() {

  doSpread();

  doRest(1,2,3,4);

  doDestructuring();

}

start();

//----------

function doSpread() {

  const marriedMen = people.results.filter(*person* => *person*.name.title == 'Mr');

  const marriedWomen = people.results.filter(*person* =>  *person*.name.title == 'Ms');

  const marriedPeople = [...marriedMen, ...marriedWomen];

  console.log(marriedPeople);

};

function doRest(...*numbers*) {

  let sum = *numbers*.reduce((*acc*, *curr*) => *acc* + *curr*, 0);

  console.log(*numbers*);

  console.log(sum);

};

function doDestructuring () {

  const first = people.results[0];

  const { username, password } = first.login;

  console.log(username, password);

  //invés de

  const usernameR = people.results[0].login.username;

  const passwordR = people.results[0].login.password;

  console.log(usernameR, passwordR);

};

## Aula 13 - Refatoração do projeto de CRUD

## Aula 14 - Introdução à programação assíncrona com JavaScript

teoria

## Aula 15 - Funções setTimeout e setInterval

* O setTimeOut() funciona para postergar uma execução de alguma coisa, ele executa o código dentro dela após tanto tempo a ser definico;
* O setTimeInterval serve para executar um código de tanto em tanto tempo;

function start() {

  const timer = document.getElementById('timer');

  const text = document.getElementById('text');

  let count = 0;

  const interval = setInterval(() => {

    timer.textContent = count++;

    if (count == 4) {

*this*.clearInterval(interval); //Para a contagem

    }

  }, 500);

  setTimeout(() => {

    text.textContent = 'setTimeOut() funcionou';

  }, 2500);

}

start();

## Aula 16 - Requisições HTTP com JavaScript

O '.fetch()' é uma função que retorna uma promise, por isso deve-se traalhar de maneira assíncrona com ele. “assim que este '.fetch()' estiver resolvido faça tal coisa”. Sendo que o ‘.catch()’ é usado quando o '.fetch()' encontra algum problema com a requisição e o ‘.then()’ quando a requisição funciona.

function start(){

  doFetch();

  divisionPromise(2,0).then((*result*) => {

    console.log(*result*);

  }).catch((*err*) => {

    console.log('este foi o erro: ' + *err*);

  })

  asyncDivisionPromise();

  asyncDoFetch();

};

start();

//----------

function doFetch() {

  fetch('https://api.github.com/users/leomac00').then((*res*) => {

*res*

      .json()

      .then((*data*) => {

        showData(*data*);

      })

      .catch((*err*) => {

        console.log(*err*);

      });

  });

}

function showData(*data*) {

  const user = document.getElementById('user');

  user.textContent = *data*.login + ' : ' + *data*.name;

}

function divisionPromise(*a*, *b*) {

  return **new** *Promise*((*resolve*, *reject*) => {

    if (*b* == 0) {

      reject('nao dá para dividir por zero');

    }

    resolve(*a* / *b*);

  });

}

async function asyncDivisionPromise() {

  const division = await  divisionPromise(2,2);

  console.log(division);

}

async function asyncDoFetch() {

  const res = await fetch('https://api.github.com/users/leomac00');

  const json = await res.json();

  console.log(json.name);

}

O ‘async’ e ‘await’ deve ser utilizado quando lidando com promises para poder facilitar o entendimento do código.

# MÓDULO 02

## Aula 01 – Apresentação do curso

## Aula 02 – Conceitos e termos

## Aula 03 - Instalação do Node.js - Visual Studio Code

Para poder utilizar o react é necessário instalar algumas dependências, sendo elas:

* NVM que pode ser baixado por este link: https://github.com/nvm-sh/nvm ;
* Node.js – utilizando o nvm, que irá gerenciar as versões do node no computador, através do comando no cmd nvm install <version>, neste caso utilizaremos a 14.15.1;

## Aula 04 - Primeira App React

Primeiramente deve-se iniciar o projeto React, para isto basta entrar no diretório em que se deseja criar o projeto pelo cmd e digitar, no exemplo criaremos o projeto ‘hello\_world’ através do npx, instalando-o utilizando comando:

* npm install -g npx

Após instalado criamos o projeto:

* npx create-react-app hello\_world

Então entrando no diretório criado é possível utilizar um comando para iniciar a aplicação;

* npm start

Aqui vale mostrar alguns comandos possíveis de serem executados ao se criar um novo app React:

* npm start: inicia o projeto em um servidor local;
* npm build: ‘constrói’ a aplicação pra poder ser enviada para a produção, ou seja, altera nomes de variáveis, exclui dependências de desenvolvimento ou desnecessárias e etc;
* npm eject: torna o gerenciamento de dependências manual;

## Aula 05 - Conhecendo nossa App

Parte 01:

A primeira coisa a se notar quando abrir a app no VSCode é o package.json, que é responsável por manipular os script que podem ser rodados no terminal e gerenciar as dependências do projeto.

As dependências do projeto ficam instaladas na pasta node\_modules dentro da pasta da aplicação.

Caso as dependências se percam ou precisem ser instaladas por alguma razão basta ir até a pasta onde está a aplicação, ou seja, onde está o package.json e rodar o seguinte comando no terminal:

* npm install

o comando irá ler o arquivo package.json e instalar todo o conteúdo necessário para rodar a aplicação. Isto pode ser usado também caso haja algum problema com alguma(s) da(s) dependências.

Parte 02:

Outro ponto a se notar nos arquivos é a pasta public, ela contém todos os arquivos responsáveis pela página, por exemplo o index.html e index.js.

Uma coisa legal de se notar aqui é que possível rodar o React em apenas uma parte da página. Para isso é necessário criar uma div onde se quer que seja executada, no caso do projeto pré-existente a div ‘root’ e então indicar que o React deve ser executado nela no index.js.

<!DOCTYPE *html*>

<html *lang*="en">

  <head>

    <meta *charset*="utf-8" />

    <link *rel*="icon" *href*="%PUBLIC\_URL%/favicon.ico" />

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1" />

    <link *rel*="apple-touch-icon" *href*="%PUBLIC\_URL%/logo192.png" />

    <link *rel*="manifest" *href*="%PUBLIC\_URL%/manifest.json" />

    <title>React App</title>

  </head>

  <body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div *id*="root"></div>

  </body>

</html>

E no js

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

ReactDOM.render(

  <*React.StrictMode*>

    <*App* />

  </*React.StrictMode*>,

  document.getElementById('root')

);

reportWebVitals();

Alguns arquivos também podem ser apagados, como é o caso das imagens que o create-react-app cria automaticamente e afins, por exemplo, só que qualquer modificação feita nestes arquivos, exclusão ou inclusão, deve ser editada no arquivo ./public/manifest.json. E praticamente a todo o conteúdo da pasta ./src pode ser apagada.

O componente React reponsável por renderizar as informações na tela é o ReactDOM, que recebe dois parâmetros:

1. O que irá renderizar;
2. Onde irá renderizar;

E para que ele funcione ele deve ser importado através do import.

## Aula 06 - Calculadora

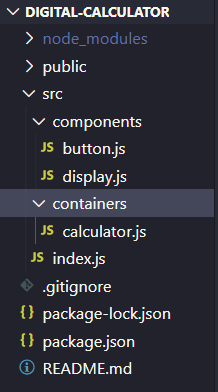
Parte 01:

A intenção desta aula é aprender o básico de React e o básico de CSS.

Aqui se faz importante freezar que se pode utilizar o React apenas em um pedacinho da página através do ReacDOM.render();

Trabalhando com React é uma boa prática criar uma pasta ./src/componentes para armazenar todos os componentes que não são containers, isto é, de forma que esses componente que fazem parte da página possam ficar melhor organizados, por exemplo um arquivo com as funcionalidades da calculadora.

De maneira análoga se cria uma pasta ./src/containers, para armazenar os container, no caso, cria-se um arquivo chamado calculator.js para armazenar a calculadora.

Outra boa prática ser observada é a de planejar antes de fazer, mesmo parecendo óbvio, costuma-se primeiro criar os arquivos e pastas necessários antes de coda-los. No exemplo, é possível observar que a calculadora possui três componentes principais: Os botões, o display e a calculadora em si. Portanto, organizam-se os elementos da seguinte maneira:

Agora dando início ao código:

Código do arquivo display.js:

//Importa do React a classe Components

import React, { Component } from 'react';

//Cria uma classe Display que é uma extensão de Components

class Display extends Component {

  //render() é a função no React.js responsável por renderizar alguma informação para o usuário

  render() {

    //no caso irá renderizar uma div com tais classes e tais valores

    return (

    <div *className*={"display borderBlack"}>

      {*this*.props.value}

    </div>

    )

  }

}

//usando 'export default' não será necessário usar '{}' na hora de importar

export default Display;

Perceba que o método render() retorna um HTML que é escrito em JSX (HTML + JSX).

Parte 02:

*“A ideia de separar em vários componentes é dividir responsabilidades.”*

Código do arquivo button.js:

import React, { Component } from 'react';

class Button extends Component {

  handleClick() {

    if (*this*.props.onClick){

*this*.props.onClick();

    }

  }

  render() {

    return(

    <div *className*={"button"} *onClick*={*this*.handleClick.bind(*this*)}>

      {*this*.props.display}

    </div>

    )

  }

}

export default Button;

Código do arquivo calculator.js:

import {Component} from 'react';

import Button from '../components/button';

import Display from '../components/display';

class Calculator extends Component {

  render() {

    return (

      <div *className*={"calculator"}>

        <*Display* *value*={"1234567890"}/>

        <*Button* *display*={"1"}/>

        <*Button* *display*={"2"}/>

        <*Button* *display*={"3"}/>

        <*Button* *display*={"4"}/>

        <*Button* *display*={"5"}/>

        <*Button* *display*={"6"}/>

        <*Button* *display*={"7"}/>

        <*Button* *display*={"8"}/>

        <*Button* *display*={"9"}/>

        <*Button* *display*={"0"}/>

        <*Button* *display*={"+"}/>

        <*Button* *display*={"-"}/>

        <*Button* *display*={"="}/>

        <*Button* *display*={"C"}/>

      </div>

    )

  }

}

export default Calculator;

Código do arquivo App.js:

import React from 'react';

import Calculator from './containers/calculator'

export function App() {

  return (

    <div *className*="scren">

      <*Calculator*/>

    </div>

  )

}

Código do arquivo index.js:

import React from 'react';

import ReactDOM from 'react-dom';

import {App} from './App';

ReactDOM.render(

  <*App*/>,

  document.getElementById('root')

);

## Aula 07 - Estilizando a nossa calculadora

Parte 01:

Para estilizar uma aplicação React trabalha-se com o CSS de maneira análoga a quando se trabalha com estilizando direto a página HTML porém já que o HTML é criado nos scripts JS deve ser levado em conta este ao invés do HTML em si, por isso é necessário importar o CSS dentro do arquivo JS que cria o HTML a ser usado.

No index.js:

import React from 'react';

import ReactDOM from 'react-dom';

import {App} from './App';

import "./index.css";

ReactDOM.render(

  <*App*/>,

  document.getElementById('root')

);

No app.js:

import React from 'react';

import Calculator from './containers/calculator';

import './app.css';

export function App() {

  return (

    <div *className*="screen">

      <div *className*="divSup"/>

      <*Calculator* />

      <div *className*="divInf"/>

    </div>

  );

}

No caso o reset do CSS foi feito no index.js e a estilização do app foi feito no app.css e app.js.

Noções de Grid no CSS:

O Grid do CSS tem por objetivo dividir a tela em frações determinadas pelo dev. No exemplo da calculadora decide-se utilizar uma divisão de linhas de ‘10fr 80fr 10fr’ onde ‘fr’ representa a fração das telas. E para fazer com que a implementação deste código CSS seja feita em toda a vista (view) é necessário determinar que o Height seja igual a 100vh (100% de view height), chegando em:

No app.css:

*.screen* {

  display: grid;

  grid-template-rows: 10fr 80fr 10fr;

  height: 100vh;

}

Parte 02:

Aqui é importate freezar que o grid-template-columns divide a tela de maneira horizontal, e o grid-template-rows de maneira vertical.

Com isso em mente utilizamos o rows e o columns para centralizar a calculadora na página.

No app.css:

*.screen* {

  display: grid;

  /\* Cria  3 linhas e 3 colunas nas frações indicadas \*/

  grid-template-rows: 10fr 80fr 10fr;

  grid-template-columns: 35fr 30fr 35fr;

  height: 100vh;

}

*.calculator* {

  /\* Insere os elementos da div 'calculator' na coluna 2, linha 2 \*/

  grid-column: 2;

  grid-row: 2;

  border: 1px solid black;

  border-radius: 5%;

  padding: 1em;

}

*.button* {

  display: grid;

  place-items: center;

  width: 3em;

  height: 3em;

  border: 1px solid black;

}

Parte 03:

Aqui consertamos a disposição dos botões na tela para que fiquem centralizados e ao usar o flex-wrap: wrap os botões quando chegarem ao final de seu container eles comecem no inicio dele na próxima linha:

*.screen* {

  display: grid;

  /\* Cria  3 linhas e 3 colunas nas frações indicadas \*/

  grid-template-rows: 10fr 80fr 10fr;

  grid-template-columns: 35fr 30fr 35fr;

  height: 100vh;

}

*.calculator* {

  /\* Insere os elementos da div 'calculator' na coluna 2, linha 2 \*/

  grid-column: 2;

  grid-row: 2;

  border: 1px solid black;

  border-radius: 5%;

  padding: 1em;

  /\* Terá altura máxima de 100% da calculadora e tudo que passar (overflow) estará escondido (hidden) \*/

  max-height: 100%;

  overflow: hidden;

  display: flex;

  flex-direction: column;

}

*.button* {

  display: grid;

  margin: 1em;

  place-items: center;

  width: 3em;

  height: 3em;

  border: 1px solid black;

  background-color: brown;

  color: white;

}

*.buttonsContainer* {

  display: flex;

  justify-content: center;

  align-items: center;

  flex-wrap: wrap;

  background-color: burlywood;

}

Parte 04:

Apenas terminamos de estilizar o código obtendo:

*.screen* {

  display: grid;

  /\* Cria  3 linhas e 3 colunas nas frações indicadas \*/

  grid-template-rows: 10fr 80fr 10fr;

  grid-template-columns: 35fr 30fr 35fr;

  height: 100vh;

}

*.calculator* {

  /\* Insere os elementos da div 'calculator' na coluna 2, linha 2 \*/

  grid-column: 2;

  grid-row: 2;

  border: 1px solid black;

  border-radius: 5%;

  padding: 1em;

  /\* Terá altura máxima de 100% da calculadora e tudo que passar (overflow) estará escondido (hidden) \*/

  max-height: 100%;

  overflow: hidden;

  display: flex;

  flex-direction: column;

  justify-content: center;

}

*.button* {

  display: grid;

  margin: 1em;

  place-items: center;

  width: 3em;

  height: 3em;

  border: 1px solid black;

  background-color: brown;

  color: white;

  cursor: pointer;

}

*.buttonsContainer* {

  display: flex;

  justify-content: center;

  align-items: center;

  flex-wrap: wrap;

  background-color: burlywood;

  border: 2px solid rgb(48, 156, 80);

}

*.display* {

  display: flex;

  justify-content: flex-end;

  border: 1px solid black;

  background-color: violet;

  color: white;

}

## Aula 08 - Dando funções à nossa Calculadora

Parte 01:

Agora voltamos a mexer com o JS do react de fato.

Toda classe em React possui um estado inicial que deve ser setada no constructor da classe.

Aqui ainda fazemos o display mostrar o valor através da função getValue() e usamos a função putValue() para pegar o valor quando clicado nos botões através do onCLick() neles.

import { Component } from 'react';

import Button from '../components/button';

import Display from '../components/display';

class Calculator extends Component {

  initialState = { firstValue: 0, secondValue: 0 };

  constructor(*props*) {

*super*(*props*); //Faz herdar o 'props' do Component

*this*.state = *this*.initialState;

  }

  putValue = (*value*) => {

    const lastValue = *this*.state.firstValue;

*this*.setState({ firstValue: lastValue \* 10 + *value* });

  };

  getValue = () => {

    return *this*.state.firstValue;

  };

  render() {

    return (

      <div *className*={'calculator'}>

        <div>

          <*Display* *value*={*this*.getValue()} />

        </div>

        <div *className*="buttonsContainer">

          <*Button* *display*={'1'} *onClick*={() => *this*.putValue(1)} />

          <*Button* *display*={'2'} *onClick*={() => *this*.putValue(2)} />

          <*Button* *display*={'3'} *onClick*={() => *this*.putValue(3)} />

          <*Button* *display*={'4'} *onClick*={() => *this*.putValue(4)} />

          <*Button* *display*={'5'} *onClick*={() => *this*.putValue(5)} />

          <*Button* *display*={'6'} *onClick*={() => *this*.putValue(6)} />

          <*Button* *display*={'7'} *onClick*={() => *this*.putValue(7)} />

          <*Button* *display*={'8'} *onClick*={() => *this*.putValue(8)} />

          <*Button* *display*={'9'} *onClick*={() => *this*.putValue(9)} />

          <*Button* *display*={'0'} *onClick*={() => *this*.putValue(0)} />

          <*Button* *display*={'+'} />

          <*Button* *display*={'-'} />

          <*Button* *display*={'='} />

          <*Button* *display*={'C'} />

        </div>

      </div>

    );

  }

}

export default Calculator;

Parte 02:

Aqui trabalhamos as operações que podem ser executadas com a calculador, utilizando o operator que será uma variável introduzida no initialState para guardar qual operação está sendo executada.

import { Component } from 'react';

import Button from '../components/button';

import Display from '../components/display';

class Calculator extends Component {

  initialState = { firstValue: 0, secondValue: 0, operator: 1, isSum: false };

  constructor(*props*) {

*super*(*props*); //Faz herdar o 'props' do Component

*this*.state = *this*.initialState;

  }

  putValue = (*value*) => {

    const {firstValue, secondValue, operator} = *this*.state;

    const lastValue =

      operator === 1

        ? firstValue

        : secondValue;

    switch (operator) {

      case 1:

*this*.setState({ firstValue: (lastValue \* 10) + *value* });

        break;

      case 2:

*this*.setState({ secondValue: (lastValue \* 10) + *value* });

        break;

    }

  };

  getValue = () => {

    const {firstValue, secondValue, operator, isSum} = *this*.state

    switch (operator) {

      case 1:

        return firstValue;

      case 2:

        return secondValue;

      case 3:

        return isSum

          ? firstValue + secondValue

          : firstValue - secondValue;

    }

  };

  pickOperation = (*isSum*) => {

*this*.setState({ operator: 2, isSum });

  };

  execOperation = () => {

*this*.setState({ operator: 3 });

  };

  clear = () => {

*this*.setState(*this*.initialState);

  }

  render() {

    return (

      <div *className*={'calculator'}>

        <div>

          <*Display* *value*={*this*.getValue()} />

        </div>

        <div *className*="buttonsContainer">

          <*Button* *display*={'1'} *onClick*={() => *this*.putValue(1)} />

          <*Button* *display*={'2'} *onClick*={() => *this*.putValue(2)} />

          <*Button* *display*={'3'} *onClick*={() => *this*.putValue(3)} />

          <*Button* *display*={'4'} *onClick*={() => *this*.putValue(4)} />

          <*Button* *display*={'5'} *onClick*={() => *this*.putValue(5)} />

          <*Button* *display*={'6'} *onClick*={() => *this*.putValue(6)} />

          <*Button* *display*={'7'} *onClick*={() => *this*.putValue(7)} />

          <*Button* *display*={'8'} *onClick*={() => *this*.putValue(8)} />

          <*Button* *display*={'9'} *onClick*={() => *this*.putValue(9)} />

          <*Button* *display*={'0'} *onClick*={() => *this*.putValue(0)} />

          <*Button* *display*={'+'} *onClick*={() => *this*.pickOperation(true)} />

          <*Button* *display*={'-'} *onClick*={() => *this*.pickOperation(false)} />

          <*Button* *display*={'='} *onClick*={() => *this*.execOperation()}/>

          <*Button* *display*={'C'} *onClick*={() => *this*.clear()}/>

        </div>

      </div>

    );

  }

}

export default Calculator;

Parte 03:

Organizamos mais ainda o código e implementamos a função de desabilitar botões através do this.props.disabled

NO button.js:

import React, { Component } from 'react';

class Button extends Component {

  handleClick() {

    const { disabled, onClick } = *this*.props;

    if (onClick && !disabled) *this*.props.onClick();

  }

  render() {

    const cssButtonClass = *this*.props.disabled ? 'button disabled' : 'button';

    return (

      <div

*className*={cssButtonClass}

*onClick*={*this*.handleClick.bind(*this*)}

      >

        {*this*.props.display}

      </div>

    );

  }

}

export default Button;

No calculator.js:

import { Component } from 'react';

import Button from '../components/button';

import Display from '../components/display';

class Calculator extends Component {

  initialState = { firstValue: 0, secondValue: 0, operator: 1, isSum: false };

  constructor(*props*) {

*super*(*props*); //Faz herdar o 'props' do Component

*this*.state = *this*.initialState;

  }

  putValue = (*value*) => {

    const {firstValue, secondValue, operator} = *this*.state;

    const lastValue =

      operator === 1

        ? firstValue

        : secondValue;

    switch (operator) {

      case 1:

*this*.setState({ firstValue: (lastValue \* 10) + *value* });

        break;

      case 2:

*this*.setState({ secondValue: (lastValue \* 10) + *value* });

        break;

    }

  };

  getValue = () => {

    const {firstValue, secondValue, operator, isSum} = *this*.state

    switch (operator) {

      case 1:

        return firstValue;

      case 2:

        return secondValue;

      case 3:

        return isSum

          ? firstValue + secondValue

          : firstValue - secondValue;

    }

  };

  pickOperation = (*isSum*) => {

*this*.setState({ operator: 2, isSum });

  };

  execOperation = () => {

*this*.setState({ operator: 3 });

  };

  clear = () => {

*this*.setState(*this*.initialState);

  }

  render() {

const {operator} = *this*.state;

    return (

      <div *className*={'calculator'}>

        <div>

          <*Display* *value*={*this*.getValue()} />

        </div>

        <div *className*="buttonsContainer">

          <*Button* *display*={'1'} *onClick*={() => *this*.putValue(1)} *disabled*={operator === 3}/>

          <*Button* *display*={'2'} *onClick*={() => *this*.putValue(2)} *disabled*={operator === 3}/>

          <*Button* *display*={'3'} *onClick*={() => *this*.putValue(3)} *disabled*={operator === 3}/>

          <*Button* *display*={'4'} *onClick*={() => *this*.putValue(4)} *disabled*={operator === 3}/>

          <*Button* *display*={'5'} *onClick*={() => *this*.putValue(5)} *disabled*={operator === 3}/>

          <*Button* *display*={'6'} *onClick*={() => *this*.putValue(6)} *disabled*={operator === 3}/>

          <*Button* *display*={'7'} *onClick*={() => *this*.putValue(7)} *disabled*={operator === 3}/>

          <*Button* *display*={'8'} *onClick*={() => *this*.putValue(8)} *disabled*={operator === 3}/>

          <*Button* *display*={'9'} *onClick*={() => *this*.putValue(9)} *disabled*={operator === 3}/>

          <*Button* *display*={'0'} *onClick*={() => *this*.putValue(0)} *disabled*={operator === 3}/>

          <*Button* *display*={'+'} *onClick*={() => *this*.pickOperation(true)} *disabled*={operator !== 1}/>

          <*Button* *display*={'-'} *onClick*={() => *this*.pickOperation(false)} *disabled*={operator !== 1}/>

          <*Button* *display*={'='} *onClick*={() => *this*.execOperation()}*disabled*={operator === 1}/>

          <*Button* *display*={'C'} *onClick*={() => *this*.clear()}/>

        </div>

      </div>

    );

  }

}

export default Calculator;

No index.css:

html,

body {

  margin: 0;

  padding: 0;

  width: 100%;

  height: 100%;

}

*.disabled* {

  background-color: gray;

}

## Aula 09 - Axios

Teórica  
Axios vs. fetch