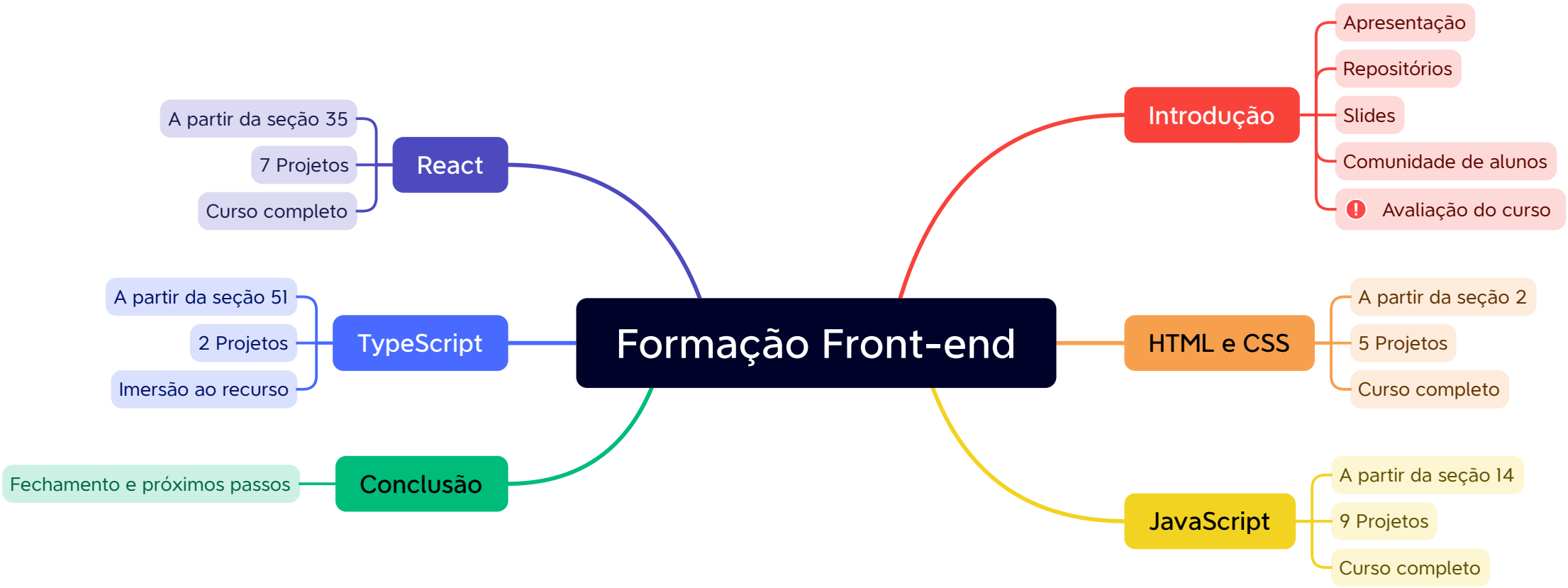


Curso completo de HTML e CSS

Crie os seus próprios sites!

O que é HTML?

- HTML não é uma linguagem de programação, e sim de marcação;
- Utilizamos para estruturar páginas web, criando elemento de texto, inserindo imagens, listas e formulários;
- É o esqueleto de qualquer aplicação web;
- Nós não precisamos de um software para compilar HTML;
- Podemos apenas abrir um arquivo .html no navegador e executar a linguagem;

O que é CSS?

- CSS é a linguagem que utilizamos para estilizar um site;
- Usamos em conjunto de HTML, a integração é super simples;
- Sem CSS todas as páginas seriam iguais, com apenas diferença no conteúdo;
- As regras de CSS são aplicadas aos elementos do HTML;
- Podemos adicionar cores, mudar o tamanho de uma fonte, adicionar bordas aos elementos e muito mais!

Editor de código

- Não precisamos de um editor para criar código de HTML e CSS;
- Porém os editores são ótimas ferramentas para programação;
- Nos ajudam com: estrutura de pastas, erros de sintaxe e highlight de sintaxe;
- Além de terem recursos extras, como terminal integrado e extensões que podem ser adicionadas;
- O meu preferido é o Visual Studio Code;

Nosso primeiro site

- Primeiramente precisamos criar um arquivo com a extensão .html;
- Podemos abrir o arquivo no navegador, sugiro o Chrome;
- Neste arquivo podemos escrever código HTML ou simplesmente texto;
- Vamos criar um e ver os resultados no navegador!

O que aconteceu?

- Nós criamos um arquivo com HTML, o navegador tem ferramentas para ler, entender e executar este tipo de arquivo;
- Ele coloca tamanhos diferentes para elementos diferentes, e os posiciona na página;
- Isso acontece porque os navegadores vêm pré-configurados para este fim;
- Vamos inspecionar o código que criamos no navegador!

Arquivos do curso

- Todos os arquivos estão no GitHub;
- É possível também fazer o download de todo o código do curso;
- Neste site podemos guardar os nossos projetos de programação de graça;
- Alguns programadores criam seu portfólio no GH, então é interessante você já criar a sua conta;
- Me segue lá também =)
- Vamos explorar o GitHub e o repositório do curso!

Curso de HTML e CSS

Fim da primeira seção

Fundamentos do HTML

Introdução

A anatomia das tags

- O elemento principal do HTML é a tag;
- Toda tag tem um nome e um propósito, a tag p serve para parágrafos;
- Nós precisamos envolver uma tag com sinais de menor e maior: assim:
- E no meio colocar um conteúdo: Texto
- Isso vai criar um elemento de parágrafo, quase todas as tags funcionam assim;

A estrutura do HTML

- Todos os projetos de HTML tem uma estrutura base, ou seja, precisamos criar algumas tags;
- **DOCTYPE**: Esta tag declara a versão do HTML;
- **html**: Esta tag envolve todo o código de HTML;
- **head**: Nesta colocamos todas as configurações de um site, como a importação de CSS e o título da página (meta tags);
- **body**: É onde todos os elementos visíveis estão;

Títulos

- Os títulos são conhecidos como headings;
- Utilizamos principalmente para separar seções;
- O nome da tag é h*, onde * pode ser um valor de 1 a 6;
- O maior título é o h1, e também é o mais importante, colocamos apenas um por página;
- A importância deve estar conectada com o propósito da nossa página, este assunto faz parte do HTML semântico, um tópico mais avançado;

Parágrafos

- Utilizamos os parágrafos para inserir textos maiores;
- A tag é <p>
- Cada parágrafo começa uma nova linha, e este comportamento acontece com todas as tags de bloco;
- Temos várias tags de bloco, os títulos fazem parte deste grupo;
- Vamos ver na prática!

Tags sem conteúdo

- Nós temos em HTML tags sem conteúdo;
- Elas possuem recursos geralmente, como quebrar linha;
- Para este fim podemos utilizar a tag
- Para uma linha horizontal temos <hr/>
- Eles também introduzem o conceito de self closing tags, onde a tag não possui uma outra tag de fechamento;
- Vamos ver na prática!

Comentários no HTML

- Comentários são utilizados para descrever como algo funciona no nosso código;
- Ou explicar a outros programadores o que fizemos;
- Os comentários não são exibidos na página;
- Mas qualquer um que inspecionar o seu código, terá acesso aos comentários;
- Vamos ver na prática!

Atributos

- Atributos podem ser utilizados para adicionar funcionalidades as tags;
- A tag a é responsável por nos direcionar a uma nova página ou site;
- Mas aonde vamos adicionar o endereço/URL?
- Colocamos no atributo chamado href;
- Um exemplo: Google
- Neste exemplo, ao clicar no link o usuário é redirecionado para o Google;

Abrir nova aba

- Com um atributo podemos fazer o link abrir em uma nova aba;
- Isso é utilizado frequentemente para redirecionar a outro site;
- Por exemplo: temos um link que leva a um e-commerce que não é nosso, não temos aquele domínio;
- Então utilizamos o atributo target com o valor _blank;
- Vamos ver na prática!

Imagens no HTML

- Nós podemos inserir imagens no nosso site com a tag img;
- O caminho relativo até a imagem é inserido no atributo src;
- Normalmente colocamos a imagem numa pasta chamada img ou assets, para fins de organização;
- Nota: a imagem é uma self closing tag;
- Vamos ver na prática!

O atributo alt

- Nas tags de imagem temos um atributo chamado alt;
- Nós inserimos nele um texto que descreve a imagem;
- Todas as imagens devem ter este atributo configurado;
- Este recurso é importante para acessibilidade, fazendo com que nosso site seja melhor rankeado pelo Google também;
- Vamos ver na prática!

Listas não ordenadas

- Listas são importantes para muitos fins nos sites;
- Podemos até criar um menu a partir de uma lista;
- As não ordenadas são criadas pela tag ul;
- Cada item na lista é representado pela tag li;
- Vamos ver na prática!

Listas ordenadas

- Listas ordenadas são interessantes para quando há um procedimento ou passos a serem seguidos;
- Exemplo: receita de algum alimento;
- Agora utilizamos a tag ol;
- E os itens continuam sendo a tag li;
- Vamos ver na prática;

Tabelas

- Nós usamos tabelas para exibir dados que podem ser categorizados em colunas;
- Tabelas são estruturas complexas no HTML, e não tão utilizadas;
- Precisamos da tag table, isso cria a tabela;
- E também um cabeçalho e um corpo;
- Cada linha é criada em uma tag tr, e os dados ficam em td;
- No cabeçalho utilizamos a tag th;

A tag div

- A tag div é utilizada para criar divisões/seções no nosso site;
- Podemos criar elementos menores também, como cards;
- O principal propósito é: encapsular elementos que estão conectados entre si;
- Vamos ver na prática!

Criando a estrutura com VS Code

- Podemos criar toda a estrutura básica do VS Code com um simples comando;
- Basta digitar ! e pressionar tab;
- Vamos ver na prática!

Desafio 1

1. Crie um novo arquivo HTML, nomeie como quiser;
2. Utilize a inicialização rápida de estrutura HTML;
3. Crie um título que descreva uma imagem;
4. Insira a imagem;
5. Crie um parágrafo que fale mais sobre o que está na imagem;
6. E por fim uma lista, com três itens com elementos presentes na imagem;

Fundamentos do HTML

Conclusão

Fundamentos do CSS

Introdução

Maneiras de adicionar CSS

- Temos algumas maneiras de adicionar CSS ao HTML;
- Inline: quando os estilos são adicionados por um atributo;
- Internal: quando o CSS é adicionado na tag head;
- External: quando o CSS é adicionado através de um arquivo externo, e depois importado no HTML;
- Vamos sempre optar pelo external CSS, quando houver a opção;
- Isso vai organizar melhor nosso projeto;

A anatomia do CSS

- Com CSS aplicamos CSS a um elemento;
- Primeiramente devemos selecionar o elemento, isso pode ser feito através da tag do elemento;
- Depois precisamos colocar as propriedades e os valores;
- Se quisermos mudar a cor de algo, utilizamos:

color: red;

- Nome da propriedade, dois pontos, valor, ponto e vírgula;

Inline CSS

- O CSS inline pode ser adicionado sem selecionar o elemento, porque é um atributo diretamente inserido no mesmo;
- O elemento já está selecionado!
- O atributo style nos permite escrever regras de CSS;
- Exemplo:

  style=”color: red;”

Múltiplas regras

- Nós podemos adicionar várias regras de CSS;
- Elas podem ser separadas por ponto e vírgula;
- Então é possível fazer uma união de estilos, dar um design melhor ao elemento;
- Vamos ver na prática!

Internal CSS

- O CSS interno é uma técnica melhor que o inline, vamos colocar todos os estilos na tag head;
- As regras precisam também estar entre a tag style;
- E através desta maneira, é necessário selecionar o elemento alvo:

```
p {
    color: red;
}
```

External CSS

- Para adicionar CSS com esta técnica precisamos criar um arquivo .css;
- Geralmente eles ficam numa pasta chamada css;
- E nós o importamos através da tag link;
- As regras que estão no arquivo são aplicadas no HTML;
- Vamos ver na prática!

Ordem do CSS

- O CSS é carregado a partir de uma ordem;
- Se temos estilos misturados (inline, internal e external), qual será aplicado?
- Todos eles, mas com a seguinte ordem: inline > internal = external > padrão do navegador;
- Esta regra funciona quando temos estilos em um mesmo elemento;
- Interno e externo tem a mesma prioridade, a última regra ganha a ‘corrida’;

Múltiplos arquivos de CSS

- É possível ter mais de uma folha de estilo no nosso projeto;
- Precisamos apenas importar todas elas na tag head;
- Os arquivos importados por último tem mais prioridade;
- Esta é uma boa prática, pois possibilita a divisão de CSS por páginas, por exemplo;
- Vamos ver na prática!

Desafio 2

1. Crie um arquivo de CSS chamado titles.css;
2. Importe este arquivo no HTML;
3. Estilize todos os h4 em alguma regra de CSS;
4. Crie um h4 no HTML e veja as mudanças de CSS;

Comentários no CSS

- Comentários do CSS são como os de HTML, utilizamos para descrever algo no código;
- Caso o código seja inspecionado, os comentários também são exibidos;
- A sintaxe é: /* algum comentário */
- Vamos ver na prática!

Classes e ids

- Classes e ids são atributos de tags do HTML, mas estão diretamente relacionados ao CSS;
- Podemos especificar elementos específicos com eles;
- Ids são utilizados para elementos únicos;
- E classes servem para um ou mais elementos, geralmente utilizadas em conjuntos de elementos;
- Veremos a utilização dos mesmos nos nossos projetos também;

Classes

- As classes são inseridas através de um atributo de HTML;
- O valor do atributo é o nome da classe, e também uma escolha nossa;
- Por exemplo: temos um botão que aparece x vezes no nosso projeto, podemos colocar uma classe btn nele;
- Ou seja, os padrões de estilo desses botões podem ser transmitidos através desta classe para os demais;
- O seletor fica: .<nome_da_classe>

IDs

- Os ids também são atributos do HTML;
- Podemos escrever qualquer coisa como valor, será o nome do id;
- Ids são únicos, ou seja, não repetimos o mesmo nome na mesma página;
- O HTML não nos proíbe disso, mas é uma má prática e deve ser evitada;
- O seletor fica: #<nome_do_id>
- Vamos ver na prática!

A ordem dos seletores

- Nós aprendemos sobre o seletor de id e classe;
- E se a tag estiver com id e uma classe, o que acontece?
- Como nas maneiras de adicionar CSS, temos também uma ordem;
- Que é: id > class > seletor de tag;
- Então o id vai vencer de todos os outros, utilize isso ao seu favor;
- Regras que não entram em conflito serão aplicadas normalmente;

As cores do CSS

- Em CSS as cores são divididas em grupos, temos:
- Nomes de cor: como red ou blue, não são muito utilizados;
- RGB: configuramos as tonalidades de red, green e blue;
- Hexadecimal: uma união de letras e números que podem criar uma cor, a maneira mais utilizada;
- HSL: hue, saturation e lightness, mudando estes valores, temos uma cor;
- Nas próximas aulas, abordaremos todos detalhadamente;

Nomes das cores

- Nós utilizamos muito essa maneira até agora, mas em projetos reais não é tão empregada;
- Pois ela nos limita a apenas as cores com nomes existentes;
- No mundo real precisamos de mais possibilidades, para não limitar os designers;
- O nome da cor consiste na utilização do nome real da cor como valor da propriedade;

HEX

- HEX ou Hexadecimal é a abordagem mais utilizada;
- Basicamente temos que inserir 6 dígitos, precedidos de uma #;
- Os dois primeiros representam o tom de vermelho, depois o de verde e por fim o azul;
- Os valores vão de 0 a 9 e A a F;
- 0 é o mais escuro e F o mais claro;
- O valor de #000 é a cor preta e #FFF a cor branca;

Mais sobre o HEX

- Se um valor for repetido 6 vezes, podemos escrever a cor de forma mais simples;
- No caso de #FFFFFF podemos reescrever com #FFF;
- A mesma coisa vale para #112233, essa cor pode ser escrita como: #123;
- Esta é uma técnica muito utilizada;
- Vamos ver na prática!

RGB

- RGB significa Red, Green e Blue;
- Nós precisamos inserir a intensidade de cada com, com valores de 0 a 255;
- 0 é o mais escuro e 255 o mais claro;
- Aplicamos RGB com a seguinte sintaxe: rgb(0-255, 0-255, 0-255)
- O primeiro valor representa o vermelho, depois verde e por fim azul;
- Para criar a cor verde inserimos: rgb(0, 255, 0);
- Vamos ver na prática!

RGBA

- Podemos criar cores também com o RGBA, A vem de alpha;
- A alteração dele muda a opacidade da cor;
- Os valores possíveis são de 0 a 1;
- Sendo 0 transparente e 1 totalmente visível;
- A sintaxe é quase a mesma: rgba(0-255, 0-255, 0-255, 0-1)
- Vamos ver na prática!

HSL

- HSL é um acrônimo para hue, saturation e lightness;
- Esta abordagem também não é muito utilizada, o ranking de uso é este: HEX > RGB > HSL > Nomes de cor;
- Podemos definir uma cor com: hsl(0-255, 0-100%, 0-100%)
- Vamos ver na prática!

Background color

- Quase todo elemento tem um background, e podemos mudar a cor dele;
- Todas as regras que vimos sobre cores podem ser aplicadas em cores de background;
- A regra é: background-color: “cor”;
- As regras de cor de fundo e cor de fonte podem ser utilizadas juntas;
- Vamos ver na prática!

Background opacity

- Podemos alterar a opacidade de uma cor de fundo com CSS;
- A regra é a opacity;
- Os valores vão de 0 a 1;
- Sendo 1 totalmente visível e 0 remove a cor;
- Com esta regra mudamos também a opacidade dos elementos dentro do elemento que alteramos a opacidade, veremos uma solução depois;
- Vamos ver na prática!

Resolvendo o problema da opacidade

- Se você não quer aplicar a opacidade para os elementos filhos, precisa utilizar o RGBA em vez de opacity;
- Alterando o valor de alpha temos a opacidade colocada apenas na cor de fundo;
- Então preservamos o conteúdo e alteramos o background;
- Vamos ver na prática!

Background images

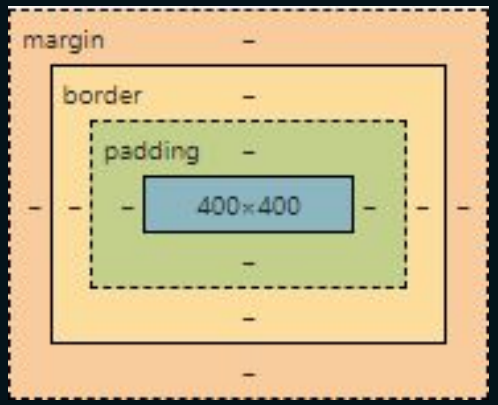
- Podemos inserir imagens no background dos elementos;
- A regra é: background-image: url(“pasta/imagem.jpg”)
- Geralmente a imagem fica em outra pasta, então temos que voltar um diretório;
- Isso pode ser feito com o símbolo ..
- Vamos ver na prática!

Centralizando a imagem de background

- Às vezes a imagem é muito mais que o elemento que estamos inserindo
- Então precisamos melhorar a visualização, centralizando a imagem;
- Isso pode ser feito com duas regras: background-position com o valor de center;
- E background-size com o valor de cover;
- Vamos ver na prática!

Box Model

- Box model é uma entidade que é criada em todo elemento do HTML;
- Ela consiste em quatro partes: altura e largura, padding, border e margin;
- Todas elas podem ser alteradas por CSS;
- Alguns elementos do HTML já vem com valor nestas regras;
- Este conceito é muito importante, veremos em detalhes nas próximas aulas;



Height and width

- A altura e a largura são o core do box model;
- Estas propriedades consistem no conteúdo do elemento;
- Podemos alterar as duas e mudar o tamanho do elemento na tela;
- Alguns elementos do HTML, os block elements, já vem com 100% de largura;
- Preenchendo a tela toda na horizontal;
- Vamos ver na prática!

Padding

- Padding é o espaço entre o conteúdo e a borda do elemento, também conhecimento como espaçamento interno;
- Este recurso é utilizado para criar uma distância entre o conteúdo (texto) e a extremidade do elemento;
- Vamos ver na prática!

Lados individuais

- Podemos adicionar o padding aos lados individuais de um elemento;
- A regra é: padding-*
- Onde * pode ser: top, left, right ou bottom;
- Assim teremos valores customizados em cada um dos lados do elemento;
- Vamos ver na prática!

Shorthand properties

- As propriedades de shorthand nos permitem adicionar padding a todas as direções com uma regra;
- Apenas precisamos utilizar a regra padding, e configurar top, right, bottom e left nesta ordem;
- Exemplo: padding: 10px 5px 12px 20px;
- Esta regra de shorthand pode ser aplicada para outras propriedades, como a margin;

Padding e width

- A padding é adicionada a largura do elemento, e isso pode ser um problema;
- Por exemplo: se precisamos seguir um layout perfeitamente;
- Um elemento com 200px de width e 25px de padding tem um tamanho total de 250px na horizontal;
- Podemos diminuir a largura do elemento, mas isso dificulta o cálculo também;
- Isso pode ser resolvido com a regra box-sizing e o valor de border-box, isso faz o elemento respeitar o tamanho que está em width;

Border

- A borda é o elemento central, fica entre padding e margin;
- Padding é o espaçamento interno e margin o externo;
- Geralmente esta regra é utilizado com propósito decorativo;
- A regra de border é definida em algumas partes: tamanho, aspecto e cor da borda;
- Vamos ver na prática!

Lados individuais da border

- Podemos adicionar borda aos lados específicos de um elemento também;
- Podemos utilizar border-*, onde * pode ser top, right, bottom e left;
- Isso é utilizado frequentemente, especialmente com a border-bottom e left;
- Vamos ver na prática!

Borda arredondada

- Com a propriedade border-radius podemos arredondar os cantos de um elemento;
- Podemos aplicá-la assim: border-radius: 5px;
- Isso faz com que o canto seja arredondado em 5px;
- Importante: Podemos arredondar elementos que não a regra de borda aplicada;
- Vamos ver na prática!

Margin

- A propriedade de margin é responsável pelo espaçamento externo do elemento;
- Podemos aplicar o recurso como aplicamos padding;
- Ou seja: lados individuais e também o shorthand;
- Vamos ver na prática!

Elementos do box model juntos

- Em alguns elementos vamos utilizar todos os recursos do box model;
- Ou seja, vamos definir o tamanho (width e height);
- Um espaçamento interno (padding);
- Decorar o elemento com bordas (border);
- Afastar o elemento de outros (margin);
- Vamos ver na prática!

Desafio 3

1. Crie um elemento que tenha todas as regras do box model;
2. Tente inserir diferentes cores, para destacar as áreas;
3. Insira um texto no elemento com a tag h2;
4. Mudar a cor de fundo do texto;
5. O elemento do box model deve ter um id, o texto uma classe;
6. Adicionar uma regra chamada outline, com o mesmo valor da propriedade border, apenas com a cor diferente;

Alinhamento de texto

- Nossos textos podem ser alinhados em várias direções;
- Por padrão ele é alinhado a esquerda;
- Porém com a regra text-align configuramos center (centro) ou right (direita), para alterar o valor default;
- Usamos muito o valor de center;
- Vamos ver na prática!

Text decoration

- Com a decoration podemos adicionar efeitos ao texto;
- É possível colocar um underline ou até mesmo uma linha que corta o texto;
- Esta regra é utilizada em casos específicos;
- A tag a tem um underline por padrão, podemos remover isso com a regra de text-decoration e o valor de none;
- Vamos ver na prática!

Transformação de texto

- Com a regra text-transform podemos alterar com o texto é exibido;
- É possível alterar para uppercase ou lowercase (letras maiúsculas ou minúsculas);
- Não há muitos valores para esta regra;
- Cuidado: O CSS deve ser aplicado quando queremos texto em uppercase, nunca escreva o texto com capslock no HTML;
- Vamos ver na prática!

Espaçamento de letras

- Com a regra letter-spacing podemos alterar o espaçamento entre letras de um texto;
- Isso é interessante em situações que o layout pede esta mudança;
- A regra é aplicada da seguinte maneira: letter-spacing: 5px;
- Vamos ver na prática!

Fontes

- Com CSS podemos alterar o tipo da fonte, com a regra font-family;
- As regras disponíveis são: Serif, Sans-serif, Monospace, Cursive e Fantasy;
- Todos os navegadores tem várias fontes que podemos utilizar;
- E ainda podemos adicionar fontes externas, por exemplo com o Google Fonts;
- Vamos ver na prática!

Estilo de fonte

- Nós podemos utilizar a propriedade font-style para mudar o aspecto das letras;
- Os valores são: normal, italic e oblique;
- Normal é o valor default, e os outros as variantes;
- Oblique é como o tipo itálico, com pequenas diferenças;
- Vamos ver na prática!

Font weight

- A propriedade font-weight pode deixar nossa fonte de texto mais fina ou grossa;
- Os valores vão de 100 a 900, sendo 100 o mais fino;
- Algumas palavras também funcionam como valores, por exemplo: bold (600);
- Existem fontes que não tem todas as variações, devemos tomar cuidado com isso;
- Vamos ver na prática!

Font size

- A regra de font-size é responsável por deixar nossa fonte grande ou pequena;
- Quanto menor o valor, menor a fonte e vice-versa;
- Esta regra é configurada com unidades de medida, como o px;
- Em CSS temos diversas unidades, veremos ao longo do curso;
- Vamos ver na prática!

Display

- Em HTML e CSS temos alguns elementos que são considerados de bloco e outros inline;
- A tag div é um exemplo de block e span um exemplo de inline;
- Com a regra display podemos mudar este comportamento, ou seja, deixar uma div como inline;
- Vamos ver na prática!

Escondendo elementos

- Existem situações que precisamos ocultar elementos;
- Basta adicionar a regra display com o valor de none;
- Então o elemento não é mais exibido, porém ainda consta no HTML;
- Vamos ver na prática!

Sobre as posições dos elementos

- A regra position e seus valores são responsáveis por posicionar o elemento na tela;
- Temos algumas possibilidades: relative, fixed, absolute, sticky e mais;
- O valor padrão é static, todo elemento começa com esta posição;
- Esta regra é essencial quando precisamos mudar onde o elemento deve ficar no projeto;
- Vamos ver em detalhes nas próximas aulas;

Position static

- O valor de static na position não faz nada com o elemento;
- Porque este é o valor padrão, ou seja, todo elemento já tem este tipo de position;
- Outros valores são afetados pelas regras top, left, right e bottom, porém static não;
- Static apenas segue o fluxo do HTML;
- Vamos ver na prática!

Position relative

- Com a position configurada como relative temos mais possibilidades que static;
- Agora as regras top, left, right e bottom, movem o elemento pela tela;
- O elemento ainda segue o fluxo do HTML;
- Atenção: normalmente não utilizamos estas regras de posição com relative;
- Vamos ver na prática!

Position absolute

- Com o valor de absolute em position, o elemento pode ser movido pela tela toda;
- Ou seja, quebramos o fluxo do HTML;
- Esta regra também é afetada por top, left, right e bottom;
- Posicionar com absolute pode ser uma solução ou um problema, dependendo do ponto de vista;
- Vamos ver na prática!

Relative com absolute

- Podemos resolver o problema de absolute com relative;
- Um elemento com position absolute é ligado ao elemento mais próximo com posição relativa, se não ele é ligado ao body;
- Isso foi o que aconteceu na aula passada;
- Então com um container com posição relative, podemos controlar melhor a área de ação dos elementos com absolute;
- Vamos ver na prática!

Position fixed

- Com fixed o elemento pode ser fixado na tela;
- Mesmo após o scroll na página, o elemento permanece na mesma posição, estando sempre presente;
- O recurso é frequentemente utilizado para criar barra de navegação fixa;
- Vamos ver na prática!

Position sticky

- Sticky também faz o elemento ficar fixo na tela;
- Mas tem um outro comportamento também, quando o elemento volta para a sua posição original ele se comporta como relative;
- A posição do elemento é onde ele foi inserido no HTML;
- Vamos ver na prática!

z-index

- Se temos dois elementos com as mesmas posições ou se eles colidem na página, podemos escolher qual será exibido;
- Utilizamos o z-index para isso;
- O elemento com maior valor prevalece;
- Vamos ver na prática!

Fundamentos do CSS

Conclusão

Formulários com HTML

Introdução

O que é um formulário?

- Forms tem o papel de receber dados do usuário e enviar para um servidor;
- Podemos validar dados;
- As tags mais utilizadas são: form, label e input;
- A tag form cria o formulário e o delimita;
- Label descreve os inputs;
- E o input é a tag que inserimos dados, temos vários tipos: number, email, text e etc;

Criando nosso formulário

- Para criar um formulário vamos precisar da tag form, que encapsula todos os elementos do formulário;
- Dentro dela temos labels e inputs, mas podemos ter outras tags como divs;
- A tag de input tem um atributo chamado type, que é onde definimos o propósito do input;
- Um input do tipo text, recebe dados de texto;
- Vamos ver na prática!

Atributos da tag form

- A tag form tem dois atributos geralmente, que são:
- action: o arquivo/página que os dados serão enviados;
- method: GET ou POST, receber dado ou enviar dado;
- Não os exploraremos em detalhes pois fazem parte do back-end, ou seja, estão fora do escopo deste curso;
- Vamos ver na prática!

Atributo name

- Utilizamos o atributo name para configurar os nossos inputs;
- O valor é ligado ao propósito do input, a sua categoria;
- Exemplo: Um input que recebe a idade de um usuário, pode ter um name com o valor de age/idade;
- Este atributo é utilizado para pegar o valor quando o form é enviado para o servidor;
- Vamos ver na prática!

Atributo da label

- A tag label tem um atributo;
- E nós o utilizamos para linkar com um input, o nome do atributo é for;
- O valor deve ser o mesmo que o atributo name do atributo que corresponde aquela label;
- Utilizamos for por propósitos semânticos;
- Isso ajuda o nosso site a ser melhor rankeado no Google;
- Vamos ver na prática!

Enviando dados de formulário

- Podemos enviar os dados do form para o servidor através de um botão de submit;
- O botão também é um input, porém mudamos o type para submit;
- Quando o usuário clicar no botão o processamento do form acontece;
- Os dados serão enviados ao servidor através de uma requisição HTTP;
- Aqui é onde precisamos de uma integração com o back-end, para aproveitar os dados do formulário;

Elemento select

- A tag select tem as opções representadas por tags de option;
- Select também tem um atributo name;
- O value estará em cada uma das options, e é isso que receberemos no lado do servidor;
- Então temos duas tags para criar um elemento de seleção: select e option;
- Vamos ver na prática!

Atributo selected

- Podemos iniciar o nosso input de select com uma opção selecionada;
- Para isso esta option precisa ter o atributo selected;
- Abordagem interessante para quando temos uma opção muito provável;
- Vamos ver na prática!

Múltiplas seleções

- Podemos criar um select que nos permite mais de uma option selecionada;
- Interessante quando queremos aceitar um ou mais dados;
- Por exemplo: opcionais de um carro;
- Precisamos apenas inserir o atributo multiple na tag select;
- Vamos ver na prática!

Elemento de textarea

- A tag textarea é semelhante ao input text;
- Podemos utilizamos para textos maiores;
- Por exemplo: A bio do Instagram;
- Isso nos permite uma área maior para digitar e verificar o texto que digitamos;
- Vamos ver na prática!

Fieldset e legend

- Fieldset é uma tag para agrupar inputs;
- E legend é como uma label, que descreve os inputs agrupados;
- Utilizamos esta tag para conectar dois ou mais inputs que tenham o mesmo sentido;
- Por exemplo: nome e sobrenome;
- Vamos ver na prática!

Datalist

- Datalist é como um select, porém com um autocomplete;
- Podemos pesquisar por possíveis valores para preencher o input;
- Ou selecionar alguma opção por meio de uma lista;
- As opções são linkadas por um atributo chamado list;
- Vamos ver na prática!

Input para senha

- Se nós estamos esperando uma senha do usuário, podemos utilizar a tag input;
- Porém no atributo type colocaremos password;
- E então o texto passa a ser exibido com *, para mascarar os dígitos;
- Vamos ver na prática!

Reiniciando o form

- Podemos reiniciar todos os campos do form;
- Isso é feito através de um input do tipo reset;
- Ele é um botão, que ao ser clicado limpa o form;
- Vamos ver na prática!

Input radio

- Este input é utilizado para selecionar apenas uma opção de várias possibilidades;
- Por exemplo: O modelo do carro que estamos comprando;
- Não podemos escolher dois, então há a necessidade da decisão entre uma das opções;
- Vamos ver na prática!

Input checkbox

- O checkbox é similar ao radio;
- Temos que selecionar uma ou mais opções, e também cancelar a seleção de uma opção;
- Por exemplo: os opcionais de um carro;
- Vamos ver na prática!

Input de data

- O input do tipo date é utilizado para selecionar uma data;
- Temos um calendário que nos auxilia para datas passadas ou futuras;
- Podemos também preencher o valor digitando;
- Vamos ver na prática!

Input para arquivos

- O input de arquivos pode ser criado com o type igual a file;
- Assim podemos enviar um arquivo ao servidor;
- Por exemplo: imagens ou pdfs;
- Vamos ver na prática!

Input para números

- Configurando o type para number temos um input que só aceita dígitos;
- Este input possui setas, que nos permite alterar o número através de cliques;
- Vamos ver na prática!

Input para e-mails

- O input de e-mail é similar ao de texto;
- Porém quando enviamos o formulário temos uma validação;
- Que checa se o texto tem o padrão de um e-mail, verificando a @, por exemplo;
- Vamos ver na prática!

O atributo value

- Com o value podemos definir um valor ao input alvo;
- Como se o usuário já tivesse preenchido algo;
- Muito utilizado quando temos um valor padrão;
- Vamos ver na prática!

Atributo disabled

- O atributo disabled é utilizado para bloquear um input;
- Então não podemos mais digitar neste input;
- Útil quando não queremos que o usuário preencha um determinado input;
- Vamos ver na prática!

Atributo placeholder

- Como atributo placeholder podemos adicionar dicas para os usuários do sistema;
- Ela será exibida no próprio input;
- Ao começarmos a preencher com algum valor, a dica some e o nosso valor que fica sendo exibido;
- Vamos ver na prática!

Atributo required

- O atributo required força o preenchimento de algum campo;
- Se tentarmos enviar o form sem um valor no campo com required, receberemos um alerta da página;
- Isto é um tipo de validação HTML;
- Vamos ver na prática!

Formulários com HTML

Conclusão

Responsividade

Introdução

O que é responsividade?

- Responsividade é a técnica de adaptar uma página web para vários dispositivos diferentes;
- Ou seja, temos mudanças de CSS baseada na resolução;
- Detectamos o que o usuário está utilizando, e adaptamos o nosso site a resolução;
- As regras de CSS são as mesmas, porém dentro de um recurso chamado media query;

Configurando a responsividade

- Primeiro vamos adicionar uma meta tag ao nosso head;
- Ela é:

  <meta name=”viewport” content=”width=device-width, initial-scale=1.0”>
- Isso faz com que o conteúdo se adapte com os dispositivos;
- Adicionamos também uma escala de 1:1;
- Vamos ver na prática!

Media query

- **Media query** é o recurso que utilizamos para criar os breakpoints;
- Os breakpoints adaptam o nosso projeto para diferentes tipos de tela;
- Configuramos uma largura, e depois as regras começam a serem alternadas dependendo do tamanho da tela;
- Vamos ver na prática!

Media query com min-width

- Se utilizarmos min-width em vez de max-width, a media query funciona ao contrário;
- Então podemos desenvolver projetos com uma técnica chamada mobile first;
- Os projetos que tem mais usuários mobile geralmente são feitos em mobile first;
- Vamos ver na prática!

Padrão de breakpoints

- Estes breakpoints são utilizados frequentemente:
- 600px e abaixo: celular;
- 768px até 600px: tablets;
- 992px até 768px: mini laptops;
- 992px e acima: laptops e desktop;
- É comum em projetos profissionais utilizarmos estes valores para desenvolvermos para diferentes dispositivos;

Adaptação para landscape

- Podemos alterar CSS apenas para usuários com visão em landscape;
- Para isso precisamos de um atributo chamado orientation com o valor de landscape, na media query;
- O código é ativado quando a orientação é alterada;
- Vamos ver na prática!

Responsividade

Conclusão

Flexbox

Introdução

O que é flexbox?

- Flexbox é um valor da propriedade display;
- Esta é a maneira mais utilizada para acondicionar elementos em um container;
- Temos diversas regras dentro do flex, veremos nas aulas seguintes;
- O flex deve ser considerado no elemento pai, e os elementos filhos serão os movimentados;
- Os elementos filhos também podem ter regras específicas;

Aplicando flex

- Para aplicar o flex vamos precisar de uma estrutura base;
- Consiste em um container e elementos filhos dentro do container;
- Colocamos a regra display com o valor de flex no container, e agora os elementos estão seguindo o fluxo do flex;
- Com flex todos os elementos se comportam parecido com os elementos inline;
- Vamos ver na prática!

Flex direction

- Podemos mudar o comportamento inline do flex (chamado de row);
- Alterando o flex-direction para column, teremos agora os itens se comportando como elementos de bloco;
- O valor padrão de direction é row;
- Vamos ver na prática!

Flex wrap

- O flex tenta colocar todos os elementos na mesma linha por padrão;
- Porém há situações que queremos x elementos por linha;
- Para isso acontecer devemos aplicar a regra flex-wrap com o valor de wrap;
- Agora as linhas respeitam a largura dos elementos;
- Vamos ver na prática!

Posicionando conteúdo

- Com justify-content é possível mudar como o conteúdo é posicionado no eixo horizontal;
- Temos alguns valores interessantes nesta propriedade;
- Por exemplo: centralizar os elementos na horizontal, basta colocar o valor de center;
- Vamos ver na prática!

Posicionamento na vertical

- Com justify-content modificamos os elementos na horizontal, já align-items nos permite mudá-los na vertical;
- A propriedade tem vários valores, como o center;
- Vamos ver na prática!

Gap

- O gap é uma regra que serve para colocar espaço entre elementos que estão no flex;
- Nós especificamos a medida em px, por exemplo;
- E este valor é adicionado entre cada um dos elementos;
- Vamos ver na prática!

Order

- Com a order podemos mudar a ordem dos elementos;
- Esta propriedade é utilizada nos elementos filhos;
- Agora começamos com as regras que são aplicadas aos elementos filhos, não ao elemento pai;
- Vamos ver na prática!

Grow

- Com a regra flex-grow podemos mudar a proporção de um ou mais elementos filhos;
- Nota: A width precisa estar sem valor, como automática;
- Exemplo: Se colocarmos grow como 2, o elemento vai crescer duas vezes mais que os outros quando estiver se adaptando no container;
- Vamos ver na prática!

Basis

- A regra flex-basis configura a largura base do elemento;
- Nós podemos trabalhar com basis e grow juntas!
- Grow vai preencher toda a largura que basis deixar vazia;
- Vamos ver na prática!

Shrink

- Shrink é o oposto de grow;
- Quando utilizamos precisamos manter o tamanho dos outros elementos, então o elemento do shrink diminui o seu tamanho para manter o dos outros;
- Nota: usamos esta regra em conjunto de basis e grow;
- Vamos ver na prática!

Flex shorthand

- Utilizando apenas flex conseguimos configurar grow, shrink e basis;
- Colocamos os valores nesta ordem também;
- Desta maneira: flex: 2 1 100px;
- Isso dá ao elemento: grow = 2, shrink = 1 e basis = 100px;
- Vamos ver na prática!

Auto alinhamento

- A regra align-self alinha um elemento diferente dos demais;
- Podemos centralizar um elemento enquanto os outros seguem a regra de alinhamento do container;
- Vamos ver na prática!

Flexbox

Conclusão

HTML Semântico

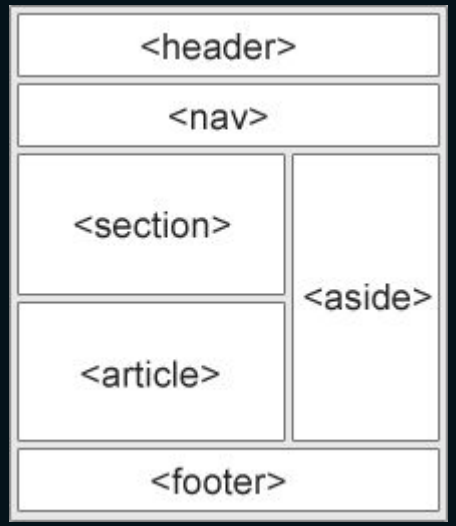
introdução da seção

O que é HTML semântico?

- Tags de HTML que dão significado ao elemento;
- Isso ajuda a estruturar melhor nosso site;
- Contribui para o SEO da nossa página;
- Exemplos de elementos não semânticos: **div**, **span**;
- Exemplos de elementos semânticos: **main**, **section**, **article**;

Estrutura da página com semântica

- Ao adicionar elementos semânticos podemos **estruturar nossa página**;
- Veja a imagem ao lado;
- **Cada seção do site** tem uma tag que faz sentido ao elemento;
- Isso até facilita a manutenção do nosso projeto;



Section

- A tag **section** define uma seção no documento;
- Geralmente um agrupamento por **categorias**, exemplo: seção dos produtos, seção de contato;
- Provavelmente um elemento que **será utilizado muitas vezes** em um projeto;
- Vamos ver na prática!

Article

- A tag **article** é utilizada para elementos informativos;
- Podemos aplicar em: post de blog, comentários, card de produtos;
- **O conteúdo é individual**, não fazendo relação com outro elemento ou article;
- Vamos ver na prática!

Header

- A tag de **header** é utilizada para um conteúdo introdução ou links de navegação;
- Geralmente tem elementos como: headings, logo, informação do autor do post/página;
- **Podemos ter mais de um header** por página;
- Vamos ver na prática!

Footer

- A tag **footer** é utilizada como rodapé de página, última seção do site;
- **Mas também pode ser utilizada em outras tags**, que precisem deste último elemento;
- Geralmente contém: informações sobre a empresa, copyright, dados de contato, sitemap;
- Vamos ver na prática!

Nav

- A tag nav é utilizada para demarcar blocos de navegação;
- Ou seja, é esperado que haja **links** dentro desta tag;
- E **nem todo link precisa estar numa nav**;
- Costumeiramente são blocos grandes de link, exemplo: barra de navegação principal;
- Vamos ver na prática!

Aside

- A tag **aside** é utilizada para criar conteúdos ao lado do conteúdo principal;
- O que fica dentro de aside **precisa estar indiretamente relacionado ao conteúdo principal**;
- Exemplo: Temos uma lista de produtos sendo exibida, em aside temos os filtros e ordenação;
- Vamos ver na prática!

Figure e Figcaption

- As tags **figure e figcaption** são utilizadas em conjunto;
- **Figure é o elemento pai**, e sua função é exibir alguma imagem na página;
- A imagem é inserida pela **tag img**;
- E figcaption serve de **legenda para a imagem**;

Main

- A **tag main** é a que contém o conteúdo principal da página;
- **Não podemos utilizar mais de uma tag main** por página;
- **O conteúdo de main deve ser único**, e não repetido ao longo da página;
- Vamos ver na prática!

Mark

- A tag **mark** deve conter um texto dentro;
- Ela define um **conteúdo que precisa estar em evidência**;
- Gerando uma importância maior para o mesmo;
- Vamos ver na prática!

HTML Semântico

Conclusão da seção

Introdução a JavaScript

introdução da seção

O que é JavaScript?

- Linguagem de programação de **alto nível**;
- Recebeu o nome por causa da **linguagem Java**, que estava na hype;
- Entenda que: JavaScript **=** JS **=** Vanilla JavaScript;
- Sua principal função é **deixar a página viva**;
- Adicionando **comportamentos** (alteração de HTML e CSS) através de **eventos**;
- JavaScript é **case sensitive**;

Onde JavaScript é utilizada?

- Interação com a página, HTML e CSS, através do **DOM**;
- Cálculo, manipulação, e validação de dados;
- Também é empregada no server-side, com **Node.js**;
- As **principais bibliotecas de Front-end** são baseadas em JS (React, Vue, Angular, Svelte...)

Formas de executar JavaScript

- Há diversas formas de executar JavaScript;
- **Padrão:** arquivo importado no HTML;
- Diretamente no navegador, através do **Console**;
- Por meio de aplicações, como o **JS Fiddle**;
- Vamos entender cada uma delas!

Repositório do curso

- O link é: **https://github.com/matheusbattisti/javascript_completo**
- Faça o download do código base;
- Você pode tirar dúvidas, consultando o código;
- Ou aproveitar os assets, como as imagens;

Introdução a JavaScript

Conclusão da seção

Tipos de dados e operadores

Introdução da seção

O que são tipos de dados?

- É a forma de **classificar um dado**;
- Temos como dado: “Matheus”, 15, true, [];
- **Os tipos de dados mais comuns são:**
  - Number;
  - String;
  - Boolean;
  - Empty values (null, undefined);
  - Object;

Number

- **Number** é o tipo de dado para valores numéricos;
- Em JS **todos os números são considerados Number**;
- Sejam eles: inteiros, ponto flutuantes ou negativos;
- Alguns exemplos: 10, 52.5, -12;
- Note que nas linguagens de programação as **casas decimais** são após o caractere ponto (15.8);
- Em JavaScript o operador typeof exibe o tipo do dado;
- Vamos ver na prática!

Aritmética com Numbers

- Podemos realizar **operações aritméticas** na programação;
- Operadores como: **+**, **-**, *****, **/**, podem ser utilizados;
- Veja um exemplo: console.log(2 + 5);
- A **ordem matemática** também é respeitada na programação, exemplo: console.log(5 + (4* 12));
- Vamos utilizá-los na prática!

Special Numbers

- **Special Numbers** são dados considerados como números, mas não funcionam como eles;
- Eles são:
  - Infinity;
  - -Infinity;
  - NaN (Not a Number);
- Algumas operações podem resultar nestes valores;
- Vamos ver na prática!

Strings

- Strings são **textos**;
- Em JavaScript temos **três formas** de criar dados de texto;
- Aspas simples, duplas e crases;
- Desta maneira: console.log(“teste”);
- **O “efeito final” é o mesmo**, mas cada um destes recursos tem particularidades;

Mais sobre strings

- Uma string deve sempre **começar e terminar com o mesmo caractere** (“, ‘, `);
- Há algumas **combinações de caracteres** que tem efeitos interessantes nas strings;
- Por exemplo o **\n**, ele pula uma linha no texto;
- Veja um exemplo: console.log(“Text em \n Duas linhas”);

Concatenação

- **Concatenação** é o recurso que une dois ou mais textos;
- O operador da concatenação é o **+**;
- Exemplo: “Meu “ + “ texto “ + “ combinado”;
- Agora o recurso pode não fazer tanto sentido, mas com variáveis teremos um melhor uso para ele;
- Vamos ver na prática!

Interpolação (Template Strings)

- A **interpolação** é um recurso semelhante a concatenação;
- Mas nos possibilita a escrever tudo na mesma string;
- Esta deve ser escrita **`entre crases`**;
- Podemos executar código JavaScript com **${ algum código }**;
- Vamos ver na prática!

Booleans

- Os booleans possuem apenas **dois valores**: true ou false;
- Qualquer comparação, utilizando os sinais >, <, ==, resulta em um booleano;
- Mais a frente veremos que este tipo é importante para **estruturas de condição e repetição**;
- Vamos ver na prática!

Comparações

- As comparações que podemos utilizar são:
- **Maior e menor:** > e <;
- Maior ou igual e menor ou igual: >= e <=;
- **Igual:** ==;
- Diferente: !=;
- **Idêntico:** ===;
- Vamos ver na prática!

Comparação de idêntico

- Os operadores **===** e **!==** funcionam como == e !==;
- Porém também levam em consideração **o tipo do dado**;
- Estes operadores necessitam que o tipo e o dado sejam iguais/diferentes;
- Devemos tentar ao máximo utilizar estes operadores;
- Vamos ver na prática!

Operadores lógicos

- Os **operadores lógicos** servem para unir duas ou mais comparações;
- O resultado final também é um boolean;
- **&&** - AND - true apenas se os dois lados forem verdadeiros;
- **||** - OR - para ser true, um lado como true é suficiente;
- **!** - NOT - este operador inverte a comparação;

Tabela verdade

- A **tabela verdade** vale para qualquer linguagem, e contém todos os resultados dos operadores lógicos;

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

Operadores lógicos na prática

- Agora vamos aplicar AND, OR e NOT na prática;
- Entender como os resultados são gerados em JavaScript;
- Este assunto é de extrema importância, faça mais exemplos para reforço;
- Vamos ver na prática!

Empty Values

- Temos duas palavras reservadas que pertencem a este grupo de dados: **undefined** e **null**;
- Undefined geralmente é visto quando utilizamos um código que ainda não foi definido;
- Já null, costuma ser imposto pelos programadores, para determinar que não há ainda um valor;
- Vamos ver na prática!

Conversão de tipo automática

- Em JavaScript algumas operações mudam o tipo de dado, e isso acontece ‘silenciosamente’;
- Exemplos:
  - 5 * null => 0
  - “5” - 3 => 2
  - “5” + 1 => 51
  - “a” * “b” => NaN
- Vamos ver na prática!

Tipos de dados e operadores

Conclusão da seção

Estruturas de programação

Introdução da seção

Salvando valores na memória

- Até então estávamos **colocando os valores nas expressões de console**;
- Porém isso não é tão comum no mundo real, nós precisamos utilizar **variáveis**;
- **Que são como containers**, que salvam informações para quando precisamos utilizar;
- Temos como declarar variáveis com let e const;
- Vamos ver na prática!

Mais sobre variáveis

- Podemos **criar várias variáveis** em sequência, desta maneira:
  - let a = 5, b = 4, c = 10
- **Não podemos** começar variáveis com números;
- Também não podemos utilizar alguns caracteres especiais, como: @;
- As variáveis são **case sensitive**;
- Vamos ver na prática!

Nomes reservados

- Algumas palavras tem o nome reservado, não podemos criar variáveis com elas, elas são:
  - break case catch class const continue debugger default delete do else enum export extends false finally for function if implements import interface in instanceof let new package private protected public return static super switch this throw true try typeof var void while with yield
- É possível unir ela mais outra palavra, para criar uma variável, ex: **let breakTeste = 1;**

O ambiente JavaScript

- Quando um programa é iniciado, um ambiente é criado;
- Neste ambiente **temos diversas funções e objetos** da linguagem JavaScript;
- Exemplo: console e alert;
- Todo programa terá acesso a elas;
- O ambiente no caso é o **navegador**;

A estrutura de uma função

- Uma função é um bloco de código que **pode ser reaproveitado ao longo do nosso programa**;
- Invocamos/chamamos ela pelo seu nome, e também o uso de parênteses: **funcao()**
- Também podemos inserir parâmetros, que deixam a execução da função única, ex: **soma(a, b)**
- Utilizamos algumas funções até então, como **log** de console;

Funções do JS: prompt

- A função **prompt** recebe um dado do usuário;
- Podemos **salvar este valor em uma variável**;
- Exemplo:
  - const x = prompt(“Digite um número:”)
- Uma função pouco utilizada, mas nos permite fazer ações interessantes;
- Vamos ver na prática!

Funções do JS: alert

- A função **alert** emite uma mensagem na tela por um pop up;
- Também não é muito utilizada, mas é um **clássico** de JavaScript;
- Vamos ver na prática!

Funções do JS: Math.x

- **Math** é um objeto, que possui diversas funções para fins matemáticos;
- Por exemplo:
  - **max**: encontra o maior número;
  - **floor**: arredonda para baixo o número;
- Vamos ver na prática!

Funções do JS: console.x

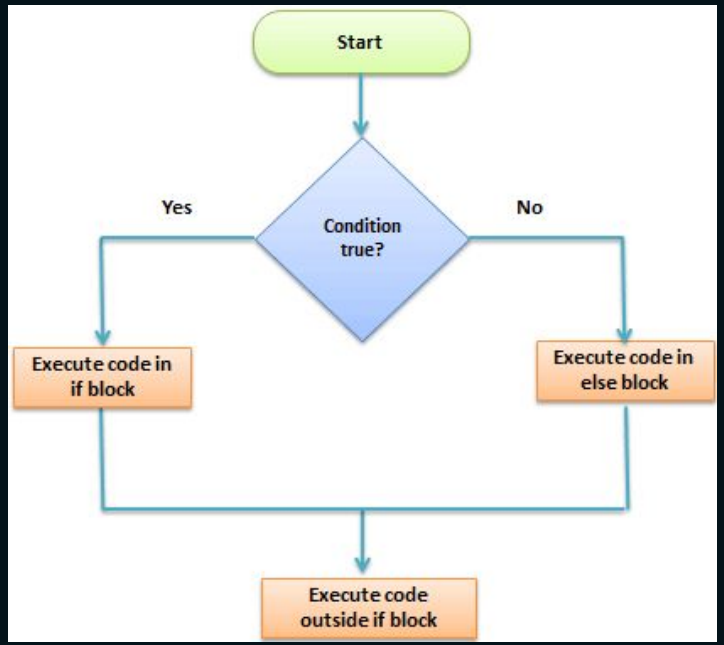
- O **console** também é um objeto, assim como Math, tem várias funções;
- A sua função principal é **exibir uma mensagem de alguma categoria** na aba de Console;
- Vamos ver na prática!

O que são estruturas de controle?

- Os programas são executados de **cima para baixo**;
- Com estas estruturas podemos **alterar o fluxo de execução**;
- O caminho dependerá das condições e comparações;
- As principais são **if e else**;

O que são estruturas de controle?

- Um exemplo de fluxo com estruturas de controle:



Estrutura condicional: if

- O **if** é muito utilizado na programação em geral;
- Temos um bloco de código sendo executado, **se uma condição for verdadeira**;
- A condição é validada por um **boolean** gerado após a execução do trecho de código no if;
- Vamos ver na prática!

Estrutura condicional: else

- O **else** executa quando o if não atende sua condição;
- Ou seja, **não temos um bloco de validação**, apenas do que será executado;
- A ideia é: Execute algo SE x > 5, SE NÃO execute isto;
- Vamos ver na prática!

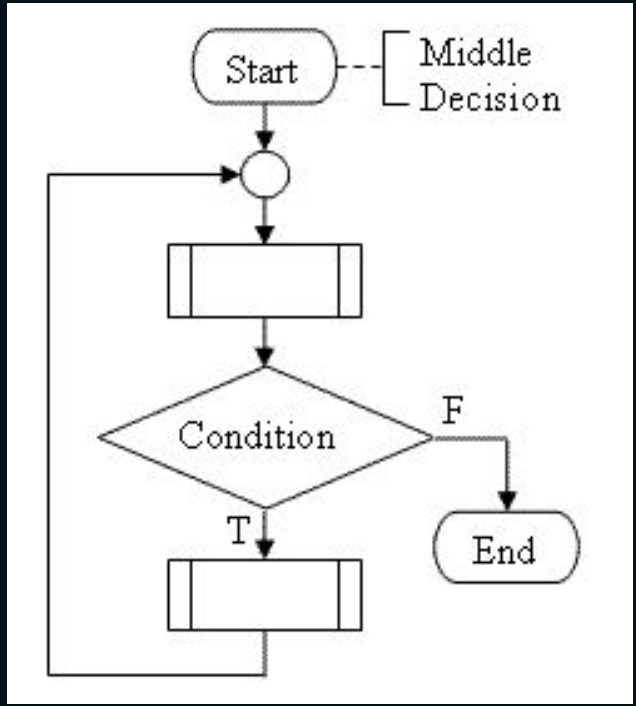
Estrutura condicional: else if

- O **else if** é uma estrutura intermediária de if e do else;
- **É possível adicionar novas condições**, como no if;
- Assim temos a possibilidade de criar várias validações, para resolver nosso problema;
- Vamos ver na prática!

O que são estruturas de repetição?

- Um bloco de código que **se repete até uma condição ser satisfeita**;
- Isso evita a **repetição desnecessária** do nosso código;
- Alguns das estruturas são: **for** e **while**;
- A suas sintaxes são diferentes, mas as duas chegam no mesmo resultado;
- Temos que nos atentar ao loop infinito;

O que são estruturas de repetição?



Estrutura de repetição: while

- O **while** faz uma ação até que uma condição seja atingida;
- No bloco definimos o fim do loop, que é a condição;
- Temos que definir também um **incrementador**, que é quem faz a condição ser atingida;
- Vamos ver na prática!

Estrutura de repetição: do while

- O **do while** também é uma estrutura que permite repetição;
- A sintaxe é **semelhante ao while**;
- Este recurso não é tão utilizado;
- Vamos ver na prática!

Estrutura de repetição: for

- O **for** é a estrutura de repetição mais utilizada;
- Ela **condensa toda lógica em uma linha**, ao primeiro olhar parece mais complexa, mas simplifica as coisas;
- Na própria declaração, colocamos: incrementador, condição final e progressão;
- Vamos ver na prática!

A importância da identação

- A **identação** é um recurso utilizado para organizar múltiplos blocos de código;
- **Utilizamos o tab** para criar um nível de identação;
- O código funciona sem, porém é interessante a adição deste recurso;
- Vamos ver na prática!

Forçando a saída de um loop

- Com a instrução de **break** podemos ejetar um loop, fazendo que com as repetições cessem;
- **Isso pode poupar memória**, pois o código será executado menos vezes;
- Não é tão comum, mas é um recurso válido da linguagem;
- Vamos ver na prática!

Pulando uma execução do loop

- A palavra reservada **continue**, pode pular uma ou mais execuções do loop;
- É um recurso utilizado de forma semelhante ao break;
- Vamos ver na prática!

Estrutura condicional: switch

- O **switch** pode ser utilizado para organização de um excesso de if/else;
- Cada if seria um **case**;
- Para cada case, temos que adicionar um **break**;
- E temos o **default**, que é como o else;
- Vamos ver na prática!

Convenção de nome de variáveis

- Nos casos abaixo temos a pior forma até a melhor, para declarar nome de variáveis:
  - let programadorcadastrado; (**ruim**)
  - let programador_cadastrado;
  - let ProgramadorCadastrado;
  - let programadorCadastrado; (**mais utilizada**)

Estruturas de programação

Introdução da seção

Funções

Introdução da seção

O que são funções?

- **Estruturas de código menores**, podemos dividir nosso código em várias funções;
- O ideal é que cada uma tenha **apenas um único objetivo**;
- Isso nos faz **poupar código**, pois podemos reaproveitá-las;
- **A linguagem tem várias funções já criadas**, e nós podemos criar as nossas;

Definindo uma função

- A estrutura da função é um pouco mais complexa;
- Primeiramente utilizamos a **palavra function**, isso inicia uma função;
- Precisamos depois **nomeá-la**;
- Os **parâmetros**, que são uma espécie de configuração, ficam entre **( )** depois do nome;
- O **corpo da função** fica entre **{ }**;
- Geralmente uma função retorna um valor;
- Vamos ver na prática;

Retorno das funções

- O retorno serve para para **processarmos um valor dentro da função** e retornar para o programa;
- A palavra reservada para este recurso é **return**;
- Se não retornamos nada a função tem utilidade, mas não externaliza o que acontece nela;
- Vamos ver na prática!

Escopo das funções

- **As funções tem um escopo separado do escopo do programa**, que é o global;
- Este escopo faz com que variáveis de fora não funcionem dentro;
- Podemos então **declarar novas variáveis**, sem interferir nas já declaradas;
- Vamos ver na prática!

Escopo aninhado (Nested Scopes)

- As formas de criar variáveis, **let e const**, nos dão a possibilidade do escopo aninhado;
- Que consiste em ter **em qualquer bloco a declaração de variáveis separadas dos outros escopos**;
- Um bloco é caracterizado por um código entre **{ }**;
- Vamos ver na prática!

Arrow function

- **Arrow function** é uma outra forma que temos de criar funções;
- É uma sintaxe resumida, que **tem algumas diferenças das funções normais**;
- Vamos ver na prática!

Mais sobre Arrow function

- A arrow function pode ter uma **sintaxe mais resumida**;
- Muito útil para **funções pequenas**;
- Onde omitimos as { } e também a instrução de return;
- Vamos ver na prática!

Argumentos opcionais

- Os argumentos/parâmetos nas funções **são obrigatórios**, precisamos passar todos;
- Porém **há casos de funções que podem funcionar sem algum dos argumentos**;
- Para resolver isso podemos fazer uma checagem do parâmetro com um if;
- Vamos ver na prática!

Argumentos com valor default

- Valor default é quando **o argumento tem um valor prévio**;
- Se for passado um novo valor, **o default é substituído**;
- Se não, o default é utilizado na função;
- Vamos ver na prática!

Closure

- **Closure** é um conjunto de funções, onde temos um reaproveitamento do escopo interno de uma função;
- Pois este escopo não pode ser acessado fora da função, já que é um bloco;
- Então há funções internas que aproveitam o escopo, e são chamadas de closure;
- Vamos ver na prática!

Mais sobre Closure

- As closures também podem servir para **salvar os resultados já executados**;
- Criando uma espécie de incrementação;
- Assim temos **uma variável que executa uma função** e modifica seu valor;
- Vamos ver na prática!

Recursão

- Um recurso que permite a função **se autoinvocar continuamente**;
- Criamos uma **espécie de loop**;
- É interessante definir uma condição final, para parar a execução;
- Vamos ver na prática!

Funções

Conclusão da seção

Arrays e objetos

Introdução da seção

Arrays

- Arrays são **listas**;
- Podemos inserir valores de qualquer tipo de dado;
- Os valores são inseridos entre **[ ]**;
- Cada valor é separado do outro por uma **vírgula**;
- Vamos ver na prática!

Propriedades

- Propriedades são como **informações de um objeto**;
- **Os arrays tem propriedades**, assim como outros tipos de dados;
- As propriedades podem ser acessadas por notação de ponto ou colchetes:
  - dado.prop ou dado[‘prop’]
- Vamos ver na prática!

Métodos

- **Métodos são como funções**, acessamos com notação de ponto e utilizamos () para invocar;
- **Um importante conceito da OOP**: Objetos são compostos por métodos e propriedades;
- Como muitos dados são objetos em JS, temos métodos e propriedades neles;
- Vamos ver na prática!

Objetos (Object Literals)

- Em JS temos um tipo de dado que é o objeto, mas seu nome técnico é **object literals**;
- **Isso porque o objeto vem da Orientação a Objetos**, com outros recursos: instância, herança...
- Já o literals possui apenas propriedades e métodos, nós mesmos os criamos;
- O objeto fica em um **bloco de { }**;
- Vamos ver na prática!

Removendo e criando novas propriedades

- Para adicionar uma nova propriedade a um objeto, utilizamos a **notação de ponto e atribuímos um valor**;
- Já para excluir, vamos utilizar o **operador delete** na propriedade alvo;
- Vamos ver na prática!

Diferença entre arrays e objetos

- **Os arrays são utilizados como listas de itens**, geralmente todos possuem o mesmo tipo;
- **Já os objetos são utilizados para descrever um item**, contém as informações do mesmo, e as propriedades possuem diferentes tipos de dados;
- Podemos ter também um **array de objetos**, isso é muito utilizado;
- Estes dois dados são muito importantes na programação;

Mais sobre Objetos

- Podemos copiar todas as propriedades de um objeto para outro com o **método assign**;
- O object literal é uma instância de um objeto, chamado **Object**;
- Um objeto ou array criado com const **pode ter seus elementos e propriedades modificados**!
- Vamos ver na prática!

Conhecendo melhor o objeto

- Podemos verificar as propriedades de um objeto pelo **método keys** de Object;
- Com o **método entries**, recebemos arrays dos nomes das propriedades com seus valores;
- Vamos ver na prática!

Mutação (Mutability)

- Outra característica interessante é a **mutação**, isso ocorre quando criamos um objeto a partir de outro;
- Este novo objeto, não é novo **e sim uma referência do primeiro**;
- As mudanças dele, podem afetar a cópia e vice-versa;
- Vamos ver na prática!

Loops em arrays

- Algo muito comum é **percorrer os arrays através de estruturas de repetição**, como for e while;
- Isso serve para utilizar o resultado de cada um dos elementos de forma simples, **sem repetição de código**;
- Vamos ver na prática!

Métodos de array: push e pop

- Os métodos de array são muito úteis para **manipular os arrays**, ou seja, alterar os seus valores de alguma forma;
- Com o **push** adicionamos um item ao fim do array;
- Com o **pop** temos a remoção de um elemento no fim do array;
- Vamos ver na prática!

Métodos de array: shift e unshift

- Ao contrário de pop e push, temos shift e unshift;
- O método **shift** remove o primeiro elemento do array;
- Já o método **unshift** adiciona itens ao início do array;
- Vamos ver na prática!

Métodos de array: indexOf e lastIndexOf

- O método **indexOf** nos permite encontrar o índice de um elemento, que passamos como argumento para o método;
- Já o **lastIndexOf** é utilizado quando há repetições de elementos e precisamos do índice da última ocorrência;
- Vamos ver na prática!

Métodos de array: slice

- O método **slice** é utilizado para extrair um array menor de um array maior;
- **O intervalo de elementos é determinado pelos parâmetros**, que são: o índice de início e o índice de fim;
- **O último elemento é ignorado**, se quisermos ele devemos somar 1 ao índice final;
- Vamos ver na prática!

Métodos de array: forEach

- O **forEach** é como uma estrutura for ou while, porém é um método;
- Ele **percorre cada um dos elementos do array**;
- Para alguns sua sintaxe pode ser mais simples;
- Vamos ver na prática!

Métodos de array: includes

- O método **includes** verifica se o array tem um elemento;
- Utilizamos no array e **como argumento colocamos o elemento que buscamos**;
- Vamos ver na prática!

Métodos de array: reverse

- O método **reverse** inverte os elementos de um array;
- Este método **modifica o array original**, então tome cuidado;
- Vamos ver na prática!

Sobre os métodos de string

- **As strings também são objetos**, ou seja, tem métodos e propriedades;
- **Alguns são muito semelhantes aos de array**;
- Note que você pode utilizar length em uma string ou em um array;
- E também acessar cada caractere pelo seu índice;
- Na próxima aula começamos os métodos de texto;

Métodos de string: trim

- O **trim** remove tudo que não é texto em uma string;
- Como: caracteres especiais e espaços em branco;
- Um método interessante para utilizar em **sanitização de dados**;
- O método não modifica o texto original;
- Vamos ver na prática!

Métodos de string: padStart

- O método **padStart** insere um texto no começo da string;
- **O texto pode ser repetido**, de acordo com o segundo argumento ao método, ele determina o máximo de caracteres do texto alvo;
- Vamos ver na prática!

Métodos de string: split

- O **split** divide uma string em um array;
- Cada elemento será determinado por um **separador em comum**;
- Os mais utilizados, são: ponto e vírgula, vírgula, espaço;
- Vamos ver na prática!

Métodos de string: join

- Já o **join** une um array em uma string;
- Podemos colocar um **separador** também, para formatar a string;
- Vamos ver na prática!

Métodos de string: repeat

- O método **repeat** repete um texto n vezes;
- Onde **n** é o número que colocamos como seu argumento;
- Vamos ver na prática!

Rest Operator / Rest Parameters

- **Rest Operator** é caracterizado pelo símbolo **...**
- Podemos utilizá-lo para receber indefinidos argumentos em uma função;
- Assim não precisamos declarar exatamente o que vamos receber, deixando a função mais ampla;
- Vamos ver na prática!

Estrutura de repetição for...of

- O **for...of** é uma estrutura de repetição semelhante ao for, porém mais simples;
- O número de repetição é **baseado no array utilizado**;
- E podemos nos referir aos elementos sem precisar acessar o índice deles;
- Vamos ver na prática!

Destructuring em objetos

- O **destructuring** é uma funcionalidade que nos permite desestruturar algum dado;
- No caso dos objetos, é possível **criar variáveis a partir das suas propriedades**, com uma simples sintaxe;
- Vamos ver na prática!

Destructuring em arrays

- O **destructuring** também pode ser utilizado para desestruturar um array em variáveis;
- A sintaxe é um pouco diferente, agora utilizaremos colchetes, e não temos nome das chaves;
- Vamos ver na prática!

JSON

- O **JSON**, JavaScript Object Notation, é um dado em formato de texto;
- Utilizamos para **comunicação entre API e front-end**;
- **Sua formatação é rigorosa**, se for mal feita o dado é invalidado e não conseguimos comunicação;
- Seu formato **lembra os object literals**;
- Regras: apenas aspas duplas e não aceita comentários;
- Vamos ver na prática!

JSON para objeto e objeto para JSON

- Na maioria das vezes vamos precisar **converter objetos para JSON**;
- **Ou um JSON para um objeto** JavaScript válido;
- Utilizamos o objeto JSON e os métodos **stringify** e **parse**;
- Vamos ver na prática!

Arrays e objetos

Conclusão da seção

Orientação a Objetos

Introdução da seção

O que é orientação a objetos?

- Um **paradigma de programação**, uma outra forma de programar;
- Utilizando **objetos** como seu principal princípio;
- A maioria dos softwares é desenvolvido neste paradigma;
- **Frameworks e bibliotecas de front-end** também são desenvolvidos com POO;
- Estávamos desenvolvendo no modo **procedural**;

Métodos

- **Métodos** podem ser adicionados aos objetos;
- **Eles são como propriedades**, mas contém uma função;
- Invocamos os métodos do mesmo modo que funções;
- Vamos ver na prática!

Aprofundando em Métodos

- Os métodos são utilizados para **interagir também com as propriedades do seu objeto**;
- Podemos exibir elas ou modificá-las;
- Podemos nos referenciar com o próprio objeto com a palavra reservada **this**;
- Vamos ver na prática!

Sobre o Prototype

- **Prototype** é um recurso que faz parte da arquitetura de JavaScript;
- **É uma espécie de herança,** onde objetos pais herdam propriedades e métodos aos filhos;
- **Por isso muitos dados são considerados objetos** e temos objetos, como: String, Number, e outros;
- Ou seja, cada dado tem um objeto pai herdou características pelo prototype;

Prototype na prática

- O recurso fundamental do prototype que temos que entender é o **fallback**;
- Quando uma propriedade não existe em um dado/objeto, **ela é procurada no seu ancestral**;
- Ou seja, é por isso que temos acesso a length em strings, por exemplo;
- Vamos ver na prática!

Mais sobre Prototype

- Quando criamos um objeto a partir de outro, este outro será o prototype do objeto criado;
- **Porém também herdará as características do objeto pai**, se for um objeto, herda de Object;
- Esta é a cadeia do prototype;
- Vamos ver na prática!

Classes básicas

- Os prototypes são originados de uma **Classe**;
- Que **é o molde dos objetos**, nela definimos os métodos e propriedades;
- **JavaScript já possui suas classes**, porém podemos criar as nossas;
- Isso é essencial para a Orientação a Objetos;
- Vamos ver na prática!

Classes baseadas em funções construtoras

- Utilizando **funções como classes**, conseguimos iniciar as propriedades com a criação do objeto;
- Chamamos de **função construtora**, este recurso;
- O construtor tem como objetivo **instanciar** um objeto, ou seja, criar um novo objeto;
- Vamos ver na prática!

Classes baseadas em funções

- Este recurso é semelhante ao anterior, mas com uma nova palavra chave: a **new**;
- Em várias linguagens o new é utilizado para instanciar novos objetos, em JS isso também acontece;
- E eles podem partir de funções;
- Vamos ver na prática!

Classes de função com métodos

- Para adicionar métodos antes da criação do objeto, **podemos acessar o prototype e colocá-los lá**;
- Esta é basicamente a essência de JavaScript;
- Porém com a evolução da linguagem, outros recursos foram criados, é o que veremos nas próximas aulas;
- Vamos ver na prática!

Classes ES6

- Nas versões mais atuais de JS abandonamos as functions e utilizamos as **classes**;
- Aqui temos recursos comuns em outras linguagens, como o **constructor**;
- Além da instância por **new**;
- Vamos ver na prática!

Mais sobre Classes

- Não podemos adicionar propriedade diretamente as classes;
- Isso precisa ser feito ao iniciá-la ou **via prototype**;
- **Métodos da classe também podem utilizar this** para se referir ao objeto instanciado;
- Os métodos não precisam da palavra function;
- Vamos ver na prática!

Override nas propriedades via Prototype

- As instâncias dos objetos são criadas baseadas nas classes;
- Ou seja, as **propriedades têm os valores definidos no construtor** ou por métodos;
- Para alterá-los podemos **utilizar o prototype**;
- Vamos ver na prática!

Symbols em Classes

- Quando utilizamos o recurso de **Symbol** com classe, é possível criar uma propriedade **única e imutável**;
- Isso é útil quando há algum dado que se repetirá em todos os objetos criados a partir da classe;
- Vamos ver na prática!

Getters e Setters

- Os **getters e setters** são bem famosos na Orientação a Objetos;
- O **get** é um método utilizado para exibir o valor de algum propriedade;
- E o **set** é utilizado para alterar o valor;
- Através de métodos, temos um bloco de código para transformação de dados;
- Vamos ver na prática!

Herança

- Uma classe pode herdar propriedades de outra por meio de **herança**;
- Utilizamos a palavra chave **extends**, para adicionar a classe que vai trazer as propriedades;
- E **super** para enviar os valores para a classe pai;
- Vamos ver na prática!

Operador instanceof

- Assim como typeof que verifica o tipo, temos o operador **instanceof**;
- Que **verifica se um objeto é pai de outro**, para ter certeza da ancestralidade;
- Isso é verificado com objeto => classe, e não através das classes;
- Vamos ver na prática!

Orientação a Objetos

Conclusão da seção

Debug e tratamento de erros

Introdução da seção

O que é bug e debug?

- **Bug**: um problema que ocorreu no código, muitas vezes por erro do programador, impede o funcionamento do software;
- **Debug**: Método de encontrar e resolver o bug, em JavaScript temos diversas estratégias para isso;
- **Validação**: Técnicas utilizadas para ter o mínimo possível de bugs no software;

Strict mode

- O **strict** é um modo de desenvolvimento que deixar o JS mais rigoroso na hora de programar;
- Deve ser declarado no **topo do arquivo ou de funções**;
- O strict não limita os recursos de JS, ele baliza a forma que você programa;
- **Bibliotecas famosas** são todas feitas em strict;

Método de debug: console.log

- O método **log** de console é muito utilizado para debug;
- Utilizamos diversas vezes nos nossos exemplos;
- Vamos ver na prática!

Método de debug: debugger

- O **debugger** é uma instrução que nos permite o debug no console do navegador;
- Podemos evidenciar os valores das variáveis em tempo real e com o programa executando, o que ajuda bastante;
- Vamos ver na prática!

Tratamento de dado por função

- **Nunca** podemos confiar no dado que é passado pelo usuário;
- **Sempre** devemos criar validações e tratamento para os mesmos;
- Ao longo do curso aprenderemos diversas técnicas;
- Vamos ver na prática!

Exceptions

- As **exceptions** são erros que nós geramos no programa;
- Este recurso faz o programa ser abortado, ou seja, **ele não continua sua execução**;
- Utilizamos a expressão **throw new Error**, com a mensagem de erro como argumento;
- Vamos ver na prática!

Try Catch

- **Try catch** é um recurso famoso nas linguagens de programação;
- Onde **tentamos executar algo em try**, e se um erro ocorrer ele **cai no bloco do catch**;
- Útil tanto para debug, como também no desenvolvimento de uma aplicação sólida;
- Vamos ver na prática!

Finally

- O **finally** é uma instrução que vai depois do bloco try catch;
- Ela é executada independente de haver algum erro ou não em try;
- Vamos ver na prática!

Assertions

- **Assertions** são quando os tratamentos de valores passados pelo usuário, geram um erro;
- Porém este recurso tem como objetivo **nos ajudar no desenvolvimento do programa**, ou seja, seria algo para os devs e não para os usuários;
- Vamos ver na prática!

Debug e tratamento de erros

Conclusão da seção

Programação assíncrona

Introdução da seção

O que é programação assíncrona?

- A programação assíncrona precisa ser utilizada quando **as respostas não são obtidas de forma imediata** no programa;
- **Chamadas a uma API** são assíncronas, não sabemos quanto tempo a resposta pode demorar;
- Até agora utilizamos só **instruções síncronas**;
- Na programação assíncrona as **execuções não ocorrem em formato de fila**, e sim no seu tempo;

Função setTimeout

- A função **setTimeout** faz parte da programação assíncrona;
- Pois estabelecemos uma ação para **ser executada após um certo tempo**;
- Ou seja, o código continua rodando e depois temos a execução da função;
- Vamos ver na prática!

Função setInterval

- A função **setInterval** é semelhante a setTimeout, ela é executada após um tempo;
- **Porém ela não para de ser executada**, temos a sua chamada definida pelo tempo de espera na execução;
- É como um loop infinito com execução de tempo controlada;
- Vamos ver na prática!

Promises

- As promises (promessas) são execuções assíncronas;
- É **literalmente uma promessa** de um valor que pode chegar em um ponto futuro;
- Utilizamos o objeto **Promise** e alguns métodos para nos auxiliar;
- Vamos ver na prática!

Falha nas Promises

- **Uma promise pode conter um erro**, ou dependendo de como o código é executado podemos receber um erro;
- Utilizamos a função **catch** para isso, podemos pegar o erro e exibir;
- Vamos ver na prática!

Rejeitando Promises

- A rejeição, **diferente do erro**, ocorre quando nós decidimos ejetar uma promise;
- Podemos fazer isso com o método **reject**;
- Vamos ver na prática!

Resolvendo várias promises

- Com o método **all** podemos executar várias promises;
- JavaScript se encarrega de verificar e retornar os seus valores finais;
- Vamos ver na prática!

Async Functions

- As **async functions** são funções que retornam Promises;
- Consequentemente há a possibilidade de receber o resultado delas depois, além da **utilização dos métodos de Promise**;
- Vamos ver na prática!

Instrução await

- A instrução **await** serve para aguardar o resultado de uma async function;
- Tornando mais simples lidar com este tipo de função, desta maneira não precisamos trabalhar diretamente com Promises;
- Vamos ver na prática!

Generators

- **Generators** funcionam de forma semelhante as promises;
- Ações podem ser pausadas e continuadas depois;
- Temos novos operadores, como: **function*** e **yield**;
- Vamos ver na prática!

Programação assíncrona

Conclusão da seção

JavaScript no navegador

Introdução da seção

Protocolos da web

- Um protocolo é uma **forma de comunicação entre computadores** através da rede;
- O **HTTP** serve para solicitar arquivos e imagens do servidor (Hyper Text Transfer Protocol);
- É possível navegar em sites através do HTTP;
- **SMTP**: protocolo para envio de email;
- **TCP**: protocolo para transferência de dados;

Conhecendo melhor as URLs

- Cada arquivo que é carregado no navegador **tem uma URL**;
- A **URL** (Uniform Resouce Locator) pode ser dividida em três partes;
- Por exemplo: https://horadecodar.com.br/index.html
- https é o **protocolo**, horadecodar.com.br é o **domínio**, que referencia um servidor (DNS > IP)
- E index.html o **arquivo/página que estamos acessando**;

Conhecendo o HTML

- **HTML** (HyperText Markup Language) é uma linguagem de marcação;
- Onde **estruturamos as páginas web**, criando elementos;
- Os elementos são chamados de tags, que podem ser: títulos, imagens, formulários, listas...
- As tags são caracterizadas por: ** **Texto** **
- Podemos adicionar estilos ao HTML com **CSS**;

A estrutura do HTML

- Toda página HTML tem duas partes importantes: head e body;
- No **head** inserimos as configurações da página, e importações de outros arquivos (CSS, JS);
- Já no **body** temos os elementos que ficam visíveis para o usuário;
- As tags possuem **atributos** que configuram os elementos;

HTML e JavaScript

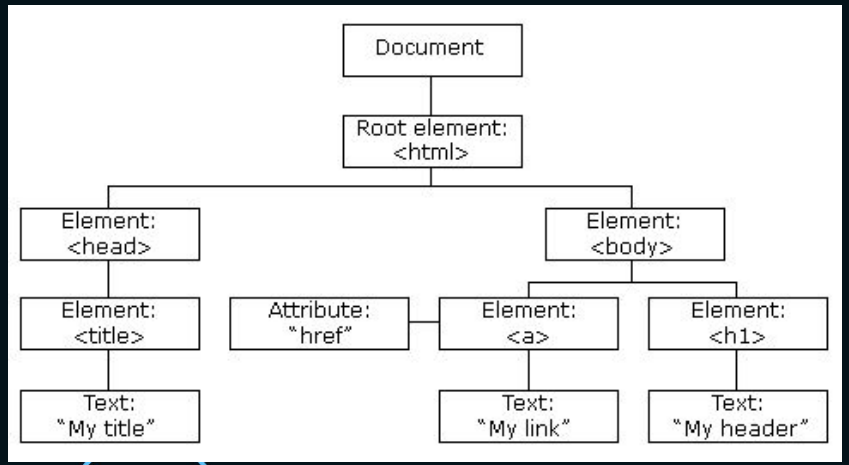
- Podemos adicionar JavaScript ao HTML por meio da **tag script**, em arquivo externo ou script na página;
- Algumas tags tem **atributos que podem executar JS,** mas isso não é muito utilizado;
- Sempre que houver um link entre um arquivo e outro, uma **chamada HTTP** é executada;
- JavaScript pode ser utilizada para **manipular elementos** do HTML e alterar estilos;

HTML e o DOM

- O **DOM** é uma representação fiel do HTML da página;
- Ele é utilizado para **acessar o HTML através de JS**, acessamos os elementos/tags;
- Assim podemos modificá-lo através dos métodos e propriedades dos objetos que alteram o DOM;
- DOM vem de **Document Object Model**;
- Através dele também podemos **atrelar eventos ao HTML**, como click ou pressionar teclas do mouse;

DOM

- O DOM pode modificar completamente uma página;
- É possível alterar: elementos, atributos, estilização;
- **Adicionamos** e **removemos** elementos;
- **O DOM cria uma árvore do HTML**, os elementos são chamados de **nós**;



Movendo-se pelo DOM

- Todos os elementos podem ser acessados através de **document.body**;
- A partir deste elemento pai, vamos encontrando os **childNodes** (nós);
- E podemos acessar suas propriedades, e consequentemente modificá-los;
- Vamos ver na prática!

Selecionando elementos

- Temos várias formas de selecionar especificamente um elemento, ou um conjunto deles;
- A diferença entre eles é a **forma de seleção**, que pode ser por: classe, id, seletor de CSS;
- Alguns exemplos são: **getElementsByTagName**, **getElementById**, **querySelector**;

Encontrando elementos por tag

- Com o método **getElementsByTagName** selecionamos um conjunto de elementos por uma tag em comum;
- O argumento é uma string que leva a tag a ser selecionada;
- Vamos ver na prática!

Encontrando elementos por id

- Com o método **getElementByID** selecionamos um único elemento, já que o id é único na página;
- O argumento é uma string que leva o id a ser selecionado;
- Vamos ver na prática!

Encontrando elementos por classe

- Com o método **getElementsByCLassName** selecionamos um conjunto de elementos por uma classe em comum;
- O argumento é uma string que leva a classe a ser selecionada;
- Veja como os atributos do HTML começam a fazer mais sentido em conjunto com JS;
- Vamos ver na prática!

Encontrando elementos por CSS

- Com o método **querySelectorAll** selecionamos um conjunto de elementos por meio de um seletor de CSS;
- E com o **querySelector** apenas um elemento, com base também um seletor de CSS;
- Vamos ver na prática!

Alterando o HTML

- Podemos mudar praticamente toda a página com DOM;
- Adicionar, remover e até clonar elementos;
- Alguns métodos muito utilizados são: **insertBefore**, **appendChild**, **replaceChild**;
- Nas próximas aulas veremos como eles funcionam;

Alterando o HTML com insertBefore

- O **insertBefore** cria um elemento antes de um outro elemento;
- É necessário criar um elemento com JS, isso pode ser feito com **createElement**;
- O elemento de referência pode ser selecionado com alguns dos métodos que vimos antes;
- Vamos ver na prática!

Alterando o HTML com appendChild

- Com o **appendChild** é possível adicionar um elemento dentro de outro;
- Este elemento adicionado será o último elemento do elemento pai;
- Vamos ver na prática!

Alterando o HTML com replaceChild

- Já o método **replaceChild** é utilizado para trocar um elemento;
- Novamente precisamos do elemento pai;
- E também o elemento para ser substituído e o que vai substituir;
- Vamos ver na prática!

Criando nós de texto

- Os textos podem ser manipulados com métodos também;
- Temos o **createTextNode**, que cria um nó de texto;
- E este nó pode ser inserido em um elemento;
- Vamos ver na prática!

Trabalhando com atributos

- Podemos ler e alterar os valores dos atributos;
- Para ler vamos utilizar o método **getAttribute**;
- E para alterar utilizamos **setAttribute**, este leva o nome do atributo e o valor para alterar;
- Vamos ver na prática!

Altura e largura dos elementos

- É possível também pegar valores com altura e largura de elementos;
- Vamos utilizar as propriedades: **offsetWidth** e **offsetHeight**;
- Se queremos desconsiderar as bordas, temos: **clientWidth** e **clientHeight**;
- Vamos ver na prática!

Posição do elemento

- Com o método **getClientBoundingRect** podemos pegar várias informações do elemento;
- Como: posição no eixo X, Y, altura, largura e outros;
- Vamos ver na prática!

Estilos com JS

- Todo elemento possui uma propriedade chamada **style**;
- A partir dela conseguimos alterar as regras de CSS;
- Note que regras separadas por traço viram camelCase, exemplo: background-color => backgroundColor;
- Vamos ver na prática!

Alterando estilos de HTMLCollection

- **HTMLCollection** aparece quando selecionamos vários elementos de uma vez;
- Podemos passar por cada um dos elementos com um for of, e estilizar individualmente cada item;
- Vamos ver na prática!

JavaScript no navegador

Introdução da seção

Eventos no JavaScript

Introdução da seção

O que são eventos?

- Ações atreladas a algum **comportamento do usuário**;
- Por exemplo: click, alguma tecla, movimento da tela e do mouse;
- Podemos inserir lógica quando estes eventos ocorrem;
- E podemos disparar eventos em certos elementos;
- Esta técnica é conhecida como **event handler**;

Como acionar um evento

- Primeiramente precisamos **selecionar o elemento** que vai disparar o evento;
- Depois vamos ativar um método chamado **addEventListener**;
- Nele declaramos qual o **tipo do evento**, e por meio de callback definimos o que acontece;
- Vamos ver na prática!

Removendo eventos

- Há situações que vamos querer remover os eventos dos elementos;
- O método para isso é **removeEventListener**;
- Passamos o evento que queremos remover como argumento;
- Vamos ver na prática!

O objeto do evento

- Todo evento possui um **argumento especial**, que contém informações do mesmo;
- Geralmente chamado de **event** ou **e**;
- Vamos ver na prática!

Propagação

- Quando um elemento de um evento não é claramente definido pode haver **propagação**;
- Ou seja, um outro elemento ativar o evento;
- Para resolver este problema temos o método **stopPropagation**;
- Vamos ver na prática!

Ações default

- Muitos elementos tem **ações padrão** no HTML;
- Como os links que nos levam a outras páginas;
- Podemos remover isso com o método **preventDefault**;
- Vamos ver na prática!

Eventos de tecla

- Os eventos de tecla mapeiam as **ações no teclado**;
- Temos a disposição **keyup** e **keydown**;
- keyup ativa quando a tecla é solta;
- E keydown quando é pressionada;
- Vamos ver na prática!

Outros eventos de mouse

- O mouse pode ativar outros eventos;
- **mousedown**: pressionou botão do mouse;
- **mouseup**: soltou botão do mouse;
- **dblclick**: clique duplo;
- Vamos ver na prática!

Movimento do mouse

- É possível ativar um evento a partir da **movimentação do mouse**;
- O evento é o **mousemove**;
- Com o objeto de evento podemos detectar a posição do ponteiro do mouse;
- Vamos ver na prática!

Eventos por scroll

- Podemos também adicionar um evento ao **scroll do mouse/página**;
- Isso é feito pelo evento **scroll**;
- Podemos determinar que algo aconteça após chegar numa posição escolhida da tela;
- Vamos ver na prática!

Eventos por foco

- O evento **focus** é disparado quando focamos em um elemento;
- Já o **blur** é quando perde o foco do elemento;
- Estes são comuns em inputs;
- Vamos ver na prática!

Eventos de carregamento de página

- Podemos adicionar um evento ao carregar a página, que é o **load**;
- E quando o usuário sai da página, que é o **beforeunload**;
- Vamos ver na prática!

Técnica de debounce

- O **debounce** é uma técnica utilizada para fazer um evento disparar menos vezes;
- Isso poupa memória do usuário, pois talvez nem sempre o evento seja necessário;
- Vamos ver na prática!

Eventos no JavaScript

Conclusão da seção

Revisão em JS Moderno

Introdução da seção

O que é JS ES6+?

- São as **novas versões** de JavaScript;
- Cada uma delas trouxe recursos que ajudam muito nós Devs;
- Estes recursos são **essenciais** para trabalhar com frameworks/libs como React, Vue e Angular;
- Agilizam muito o desenvolvimento com JS;

Variáveis com let e const

- Temos duas novas formas de declarar variáveis a partir do ES6, que são **let** e **const**;
- **let** é uma forma de atribuir valor, e poder modificar depois;
- Já **const** declara uma constante, podemos atribuir um valor e não alterar;
- O grande diferencial são os escopos em bloco;
- Vamos ver na prática!

Arrow functions

- As **arrow functions** são um recurso para criar funções de forma mais simples;
- Alguns aspectos a diferenciam das funções comuns;
- Por exemplo o this, que é relacionado ao elemento pai de quem está executando;
- Vamos ver na prática!

Filter

- O **filter** é um método de array para filtrar dados;
- O filtro é determinado por nós, **por meio de uma função**;
- Resultado em um array com **apenas os elementos que precisamos**;
- Nessas versões mais novas de JS temos vários métodos de array importantes como este;
- Vamos ver na prática!

Map

- O **map** também é um método de array, percorre todos os elementos do mesmo;
- O map é utilizado para **modificar o array de origem**;
- Filter remove elementos desnecessários, map altera os que precisamos;
- Vamos ver na prática!

Template literals

- O recurso de **template literals** permite a impressão de variáveis em um texto;
- Escrevemos entre crases, desta maneira: `texto`
- E as variáveis são inseridas com: **${**variavel**}**
- Vamos ver na prática!

Destructuring

- O **destructuring** desestrutura dados complexos em várias variáveis;
- Podemos utilizar em **arrays e objetos**;
- Muitas variáveis podem ser criadas em um única linha;
- Vamos ver na prática!

Spread operator

- O **spread** pode ser utilizado em **arrays e objetos**;
- Utilizamos para inserir novos valores em um array ou objeto;
- É um recurso que pode unir dois arrays, por exemplo;
- Vamos ver na prática!

Classes

- As **classes** são recursos fundamentais para programar orientado a objetos;
- Temos acesso a recursos importantes, como: constructor, propriedades, métodos;
- Antes as classes em JS eram criadas com **constructor functions**;
- Vamos ver na prática!

Herança

- **Herança** é o recurso que nos dá a possibilidade de uma classe herdar métodos e propriedades de outra;
- A palavra **extends** determina qual classe será herdada;
- Para enviar propriedades para a classe pai utilizamos **super**, isso é necessário;
- Vamos ver na prática!

Revisão em JS Moderno

Conclusão da seção

Axios

Introdução da seção

O que é Axios?

- Uma biblioteca JavaScript para **requisições HTTP**;
- Axios é **Promise based**, ou seja, retorna promessas de suas funções;
- Torna muito mais simples o trabalho com APIs e requisições assíncronas;
- Muito utilizado nas empresas;
- Apesar disso, perdeu muita notoriedade para o recurso de **fetch** da JS;

Instalando o Axios

- Para instalar o Axios basta **copiar um link de script externo** para o nosso projeto;
- O link da documentação é: https://axios-http.com/
- Em projetos que utilizamos bibliotecas e frameworks, utilizamos o **npm** para instalar o Axios;
- Vamos ver na prática!

Nosso primeiro request

- Para fazer uma requisição podemos utilizar o método **get**, isso vai nos trazer dados de algum local;
- É recomendado utilizar um **try catch** para identificar possíveis erros;
- Como o Axios é baseado em promises, podemos utilizar as **async functions**;
- Vamos ver na prática!

Exibindo os dados na tela

- Após um request é comum **exibir os dados na tela**;
- Podemos fazer isso juntando a resposta da chamada com os nossos conhecimentos em **DOM**;
- Criar elementos baseado no que veio na requisição;
- Vamos ver na prática!

Configurando os headers

- Os **headers** são configurados no momento da requisição;
- Podemos adicionar **parâmetros adicionais**;
- Por exemplo: determinar o tipo de dado que queremos;
- Vamos ver na prática!

Requisição de POST

- Para enviar dados vamos utilizar o método **post**;
- É necessário configurar a **propriedade body** com os dados a serem enviados;
- Vamos ver na prática!

Global Instance do Axios

- Podemos alterar diretamente as **configurações do Axios**;
- Isso nos gera uma facilidade de trabalhar com os **mesmos parâmetros em todas as requisições**;
- Ou seja, se configuramos os headers na Global, não é necessário configurar nas requisições;
- Vamos ver na prática!

Custom Instance do Axios

- A **Custom Instance** é semelhante a Global Instance;
- Porém aqui temos outras propriedades que são possíveis de configurar, como a **baseURL**;
- Esta estratégia deve ser utilizada para personalização do nosso projeto;
- Obs: não é recomendado utilizar as duas instances juntas (manutenção, configuração em dois locais...);
- Vamos ver na prática!

Interceptors

- Interceptors são como **middlewares**;
- Ou seja, podemos **interceptar a requisição e a resposta**;
- Inserindo algum código entre estas duas ações;
- Vamos ver na prática!

Axios

Conclusão da seção

Axios x Fetch

Diferenças e formas de aplicar

O que é Fetch?

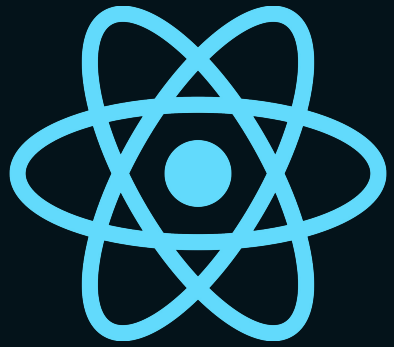
- É uma API que permite que um aplicativo faça **requisições HTTP** para recuperar recursos da web.
- É suportado por quase todos os navegadores modernos.
- Alternativa ao método **XMLHttpRequest (XHR)** tradicional, que foi usado anteriormente para fazer requisições HTTP.

O que é Axios?

- É uma **biblioteca JavaScript** que permite fazer solicitações HTTP facilmente.
- **Baseado na Fetch API**, mas oferece algumas vantagens adicionais, como a possibilidade de cancelar solicitações e tratar erros de maneira mais fácil.
- Compatível com navegadores modernos e também pode ser usado em **aplicativos Node.js**.

Axios x Fetch

- A Fetch API pode ser um pouco mais difícil de usar do que o Axios, pois **requer mais código para tratar erros** e cancelar solicitações.
- O Axios oferece uma API mais simples e intuitiva, com **métodos mais fáceis de usar** para fazer solicitações e tratar erros.
- O Fetch já vem por padrão, **o Axios precisamos instalar como dependência**;

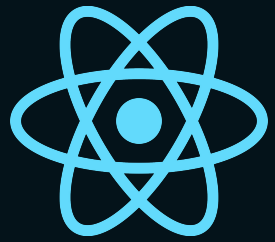


Introdução a React.js

introdução da seção

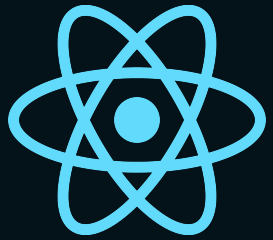
O que é React?

- React é uma **biblioteca JavaScript** para desenvolvimento de aplicações front-end;
- A categoria delas é **SPA** (Single Page Application);
- Podemos criar uma aplicação com React, ou inserir em um projeto já em andamento;
- A sua arquitetura é baseada em **componentes**;
- É mantido pelo **Facebook/Meta**;



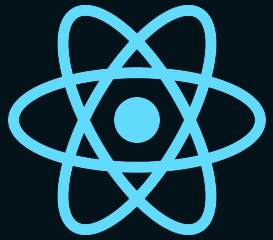
Pré requisitos para rodar React

- Para iniciar uma aplicação React da maneira convencional precisaremos de **Node.js**;
- Através do gerenciador de pacotes **npm**, é possível iniciar projetos;
- Vamos ver como instalar o Node!



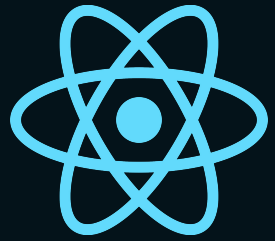
Hello World em React

- Para criar nossas aplicações utilizaremos o **Vite**;
- Antigamente era muito comum utilizar o **create-react-app**, porém ele tem uma pior performance;
- Apenas precisamos digitar no terminal: **npm create vite@latest** e seguir as opções;
- Vamos ver na prática!



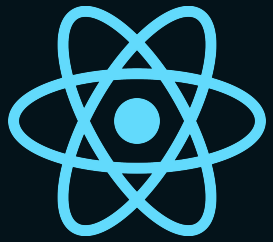
Estrutura base do React

- Há algumas pastas e arquivos muito importantes;
- **node_modules**: dependências do projeto;
- **public**: assets e arquivos estáticos;
- **src**: onde escrevemos o código da aplicação;
- **src/index.js**: arquivo de inicialização da aplicação;
- **src/App.js**: componente principal inicial (pode ser modificado);



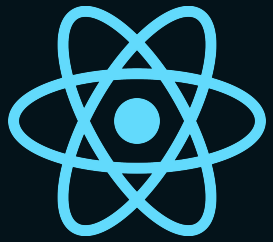
Extensão para React

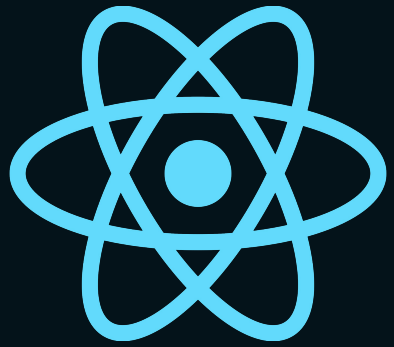
- Há diversas **extensões** interessantes para React no VS Code;
- A principal e mais utilizada é a: **ES7 React snippets**;
- Ela ajuda a criar rapidamente estruturas que utilizamos em todo projeto;
- Vamos baixar!



Preparando o Emmet para o React

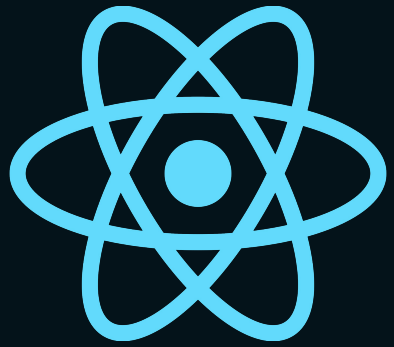
- O **Emmet** é uma extensão nativa do VS Code que ajuda a escrever código mais rápido;
- Mas ela não vem configurada para o React;
- Vamos acessar: **File > Settings > Extensions** e procurar o Emmet;
- Lá precisamos incluir: **javascript - javascriptreact**;
- Vamos configurar!





Introdução a React.js

Conclusão da seção

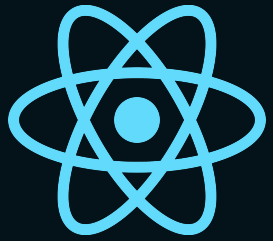


Fundamentos do React.js

Introdução da seção

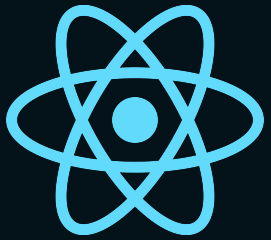
Criando componentes

- Os componentes ficam dentro de uma pasta chamada **components**, que criamos em src;
- Nomeados em CamelCase: **FirstComponent.jsx**;
- A utilização da extensão **.jsx** facilita a formatação para os editores;
- Dentro do componente precisamos **criar e exportar uma função**, que é a lógica dele;
- Vamos ver na prática!



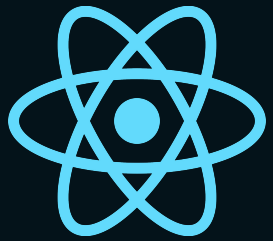
Importando componentes

- Para utilizar e reutilizar um componente é necessário o processo de **importação**;
- A sintaxe é: **import X from ‘./components/X.jsx’** onde X é o nome do componente;
- Para inserir o componente dentro de outro vamos utilizar a sintaxe de tag do HTML com o nome do componente: **<FirstComponent />**
- Vamos ver na prática!



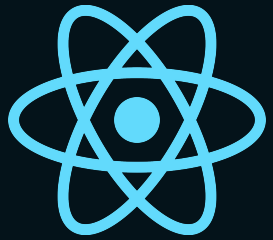
JSX

- **JSX é o HTML do React**, o código interno das funções de componentes, após o return;
- Vamos escrever as nossas tags e importar os outros componentes;
- Há algumas diferenças do HTML, ex: **class = className**;
- Podemos **escrever JavaScript dentro do JSX**;
- O JSX pode ter **apenas um elemento pai**;



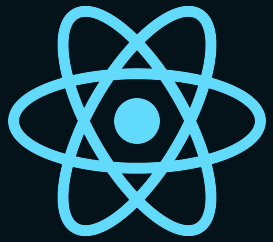
Comentários no Componente

- Há duas formas de inserir comentários em React;
- Podemos utilizar a sintaxe de JS fora e dentro das funções, com: **// Comentário**
- Ou no JSX com: **{ /* Algum comentário */}**
- As chaves são necessárias para executar qualquer instrução de JS;
- Vamos ver na prática!



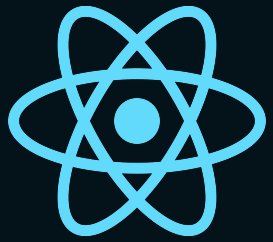
Template Expressions

- **Template Expression** é o recurso que permite a execução de JS no JSX;
- Podemos também inserir variáveis;
- A sintaxe é: **{ 2 + 2 }**
- Tudo que vai entre as chaves é entendido e executado como JavaScript;
- Vamos ver na prática!



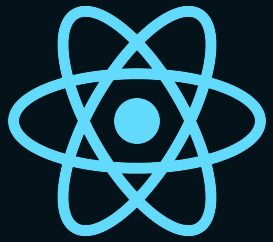
Hierarquia de componentes

- Os componentes podem ser **reutilizados** em outros componentes;
- Podemos montar também uma **hierarquia**, onde um componente é pai do outro;
- E ao importar o componente pai, todos os outros vem juntos;
- Vamos ver na prática!



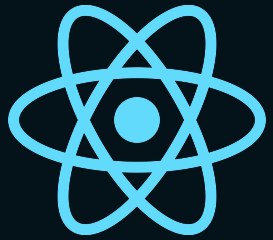
Evento de click

- Os eventos são essenciais para programar apps de front-end, vimos isso em **DOM**;
- **Em React temos os mesmos eventos**, só que de forma simplificada;
- Por exemplo: com **onClick**, conseguimos disparar um evento que ativa uma função ao clicar em um elemento;
- Vamos ver na prática!



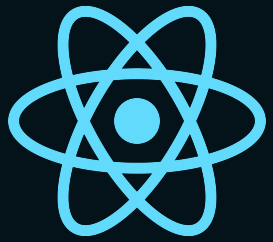
Funções nos eventos

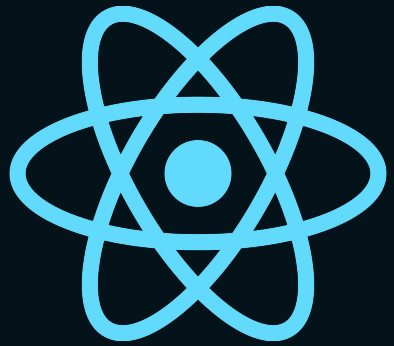
- Quando temos **lógicas complexas**, é mais indicado criar uma função para o evento;
- Isso vai separar as responsabilidades, e deixar nosso código mais de dar **manutenção**;
- Vamos ver na prática!



Funções de renderização

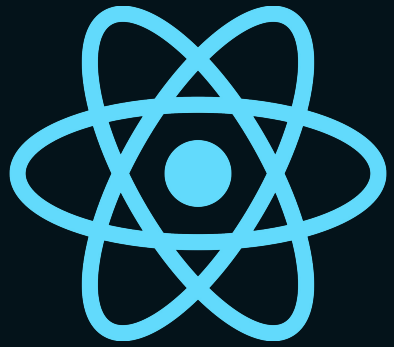
- Podemos criar funções nos componentes que **retornam JSX**;
- Isso pode ser utilizada para uma **renderização condicional**, por exemplo;
- Fazendo que o JSX varie dependendo do resultado da operação;
- Vamos ver na prática!





Fundamentos do React.js

Conclusão da seção

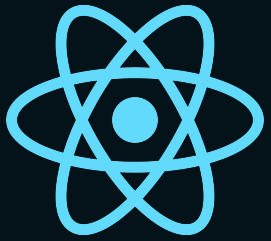


Avançando no React.js

Introdução da seção

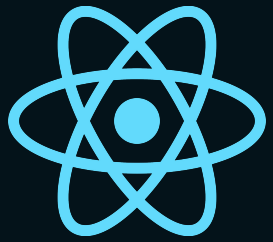
Imagens no React

- As imagens do projeto podem ficar na pasta **public**;
- Estando lá, elas **podem ser utilizadas diretamente no projeto**;
- A pasta public fica linkada com a src, exemplo: **“/imagem.png”**
- Vamos ver na prática!



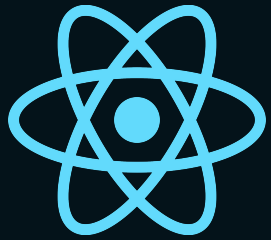
Imagens em assets

- Outro local comum de colocar as imagens em um projeto em React é na pasta **assets**;
- Em assets **precisamos importar a imagem**, como se fosse um componente;
- Estas duas abordagens são muito utilizadas;
- Vamos ver na prática!



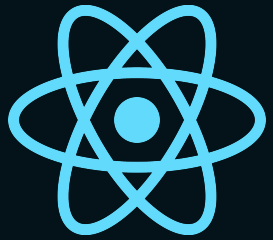
O que são hooks?

- Recursos do React que tem diversas funções, podemos criar os nossos também;
- Exemplo: **guardar e alterar o estado de algum dado**;
- **Os hooks precisam ser importados**, e sempre começam com a palavra **use**;
- Alguns bem utilizados são: **useState**, **useEffect**;
- Os hooks que nós criamos são chamados de custom hooks;
- Geralmente toda a aplicação usa pelo menos um hook;



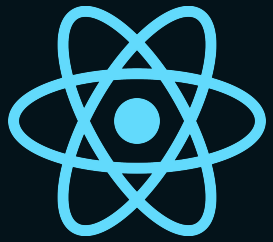
hook: useState

- O **useState** é um dos mais utilizados;
- Podemos **gerenciar o estado de um ou mais dados**, é como se fosse um getter/setter;
- Utilizamos este hook pois as **variáveis não funcionam como esperado**, elas não re-renderizam o componente;
- Para guardar um dado vamos utilizar **setNomeDoDado**;
- Vamos ver na prática!



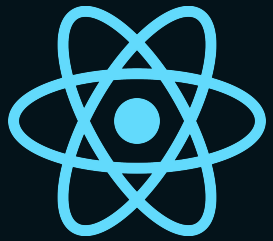
Renderização de lista

- Dados do **tipo array** são muito comuns em aplicações;
- Geralmente recebemos um **array de objetos**, e precisamos iterar nele e exibir os elementos;
- O método **map** fará a iteração;
- É possível inserir **JSX na execução**;
- Vamos ver na prática!



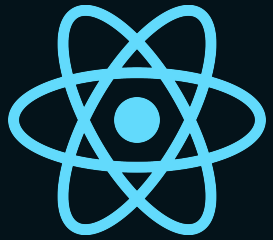
A propriedade key

- Iterar uma lista sem a **propriedade key**, gera um erro no console;
- O React precisa de uma **chave única** para cada elemento;
- Isso serve para ajudar a **renderização do componente**;
- O React utiliza isso para manipulação dos itens;
- Vamos ver na prática!



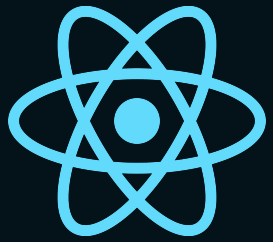
Previous state

- **Previous state** é um recurso do hook useState;
- Podemos pegar o **valor original dos dados**, e fazer alguma alteração;
- Muito utilizado em listas, pois pegamos o valor antigo e o modificamos;
- **O primeiro argumento do set** sempre é o previous state;
- Vamos ver na prática!



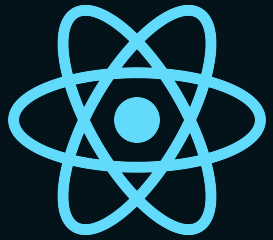
Renderização condicional

- **Renderização condicional** é quando parte do template é exibido por meio de uma condição;
- Que é simplesmente um **if no JSX**;
- Utilização: quando usuário está autenticado/não autenticado;
- Vamos ver na prática!



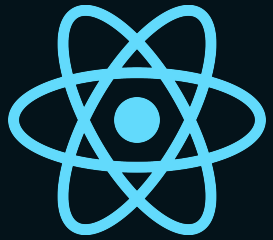
Adicionando um else

- A renderização condicional pode conter um **else** também;
- A estrutura é igual a do **if ternário**;
- Fica desta forma: condição ? execução1 : execução2
- Vamos ver na prática!



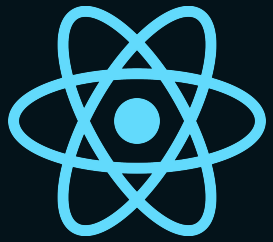
Props

- **Props** é um recurso fundamental do React;
- Permite a **passagem de dados** de um componente pai para um componente filho;
- Será útil para quando houver dados vindo de um banco de dados;
- As props vem em um **objeto no argumento da função** do componente;
- Vamos ver na prática!



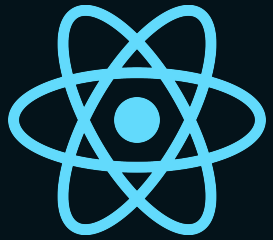
Desestruturando props

- Os componentes geralmente tem mais de uma props;
- Para facilitar o uso delas, podemos **desestruturar no parâmetro da função** do componente;
- Assim o objeto props vira o nome de cada propriedade, então não precisamos acessá-lo;
- Desta maneira: **MyComponent({name, age})**
- Utilizamos então name, em vez de props.age;
- Vamos ver na prática!



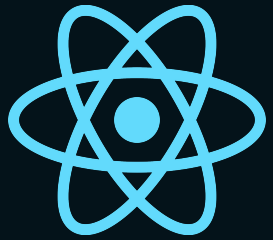
Reutilização de componentes

- Com o auxílio das props, **reutilizar componentes faz mais sentido**;
- Se temos 1000 dados de carros, **podemos representá-los com apenas um componente** repetido n vezes;
- Isso torna o código padronizado, e facilita a manutenção;
- Vamos ver na prática!



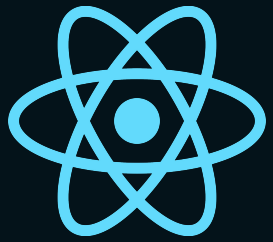
Reutilização com loop

- Os arrays podem ter muitos itens, e as vezes nem sabemos ao certo sua quantidade total;
- Então o correto é utilizar uma **estrutura de loop**, para poder percorrer os itens;
- Com isso conciliamos alguns conceitos aprendidos: **renderização de lista**, **reaproveitamento de componentes** e **props**;
- Vamos ver na prática!



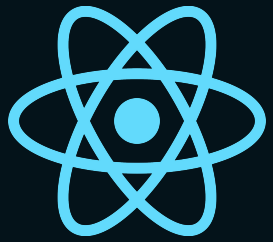
React Fragments

- Os **Fragments** são interessantes para quando há mais de um elemento pai no componente;
- Ou não queremos incluir HTML desnecessário no elemento pai, **não alterando sua estrutura**;
- A sintaxe é: **<> ... </>**
- Vamos ver na prática!



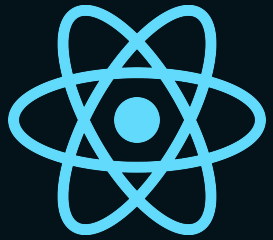
Children prop

- **Children prop** é utilizado quando um componente precisa ter JSX dentro dele;
- Porém o JSX vem do componente pai;
- Então o componente age como um **container**, abrigando esse JSX;
- E children entra como uma **prop do componente**;
- Vamos ver na prática!



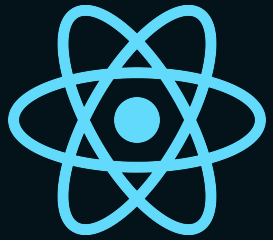
Funções em prop

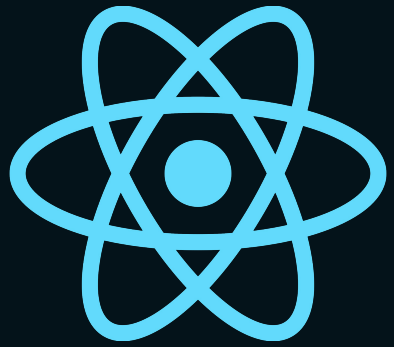
- Podemos passar **funções através de props**;
- Basta criar a função no componente pai, e enviar como prop;
- No componente filho, podemos utilizar para algum **evento**;
- Vamos ver na prática!



Elevação de state

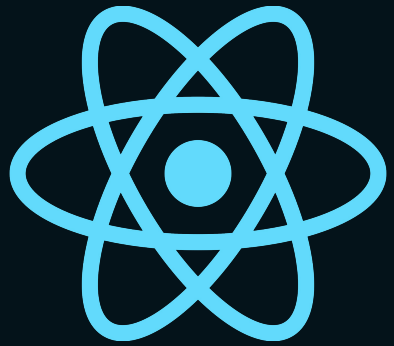
- Elevação de state ou **state lift**, é quando um valor é levado do componente filho para o pai;
- Geralmente temos um componente que usa o state e outro que o altera;
- Então **o componente pai vai gerenciar os valores** e passar para os filhos as alterações;
- Vamos ver na prática!





Avançando no React.js

Conclusão da seção

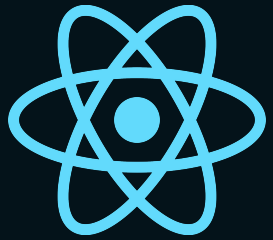


CSS no React.js

Introdução da seção

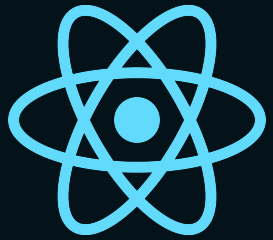
CSS Global

- O CSS global é utilizado para **aplicar estilos a todos elementos** do projeto;
- Utilizamos o arquivo **index.css** para isso, ele está na pasta src;
- Vamos ver na prática!



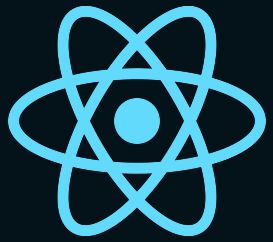
CSS de Componente

- O **CSS de componente** é utilizado apenas em um componente específico;
- Geralmente o arquivo é criado com o **mesmo nome do componente**;
- Lembre-se: ele **não é scoped**, ou seja, pode vazar para outros elementos do projeto;
- Vamos ver na prática!



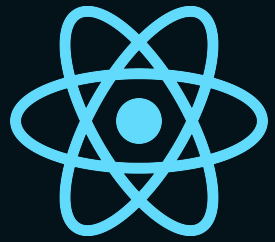
Inline style

- O **inline style** do React é igual ao do CSS;
- Por meio do **atributo style**, aplicamos regras de CSS diretamente a um elemento;
- As outras abordagens são mais interessantes que essa por questões de manutenção do código;
- Vamos ver na prática!



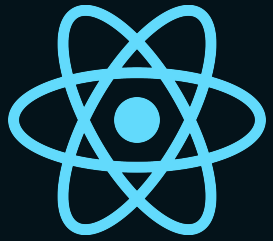
Inline style Dinâmico

- O **CSS dinâmico inline** consiste em uma técnica de aplicação de estilo por condição;
- Teremos o atributo inserido em um **if ternário**;
- Dependendo da condição e do resultado dela, um estilo diferente pode ser exibido;
- Vamos ver na prática!



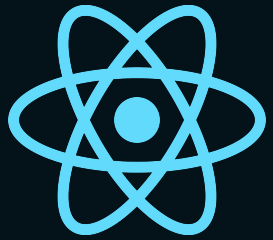
Classes dinâmicas

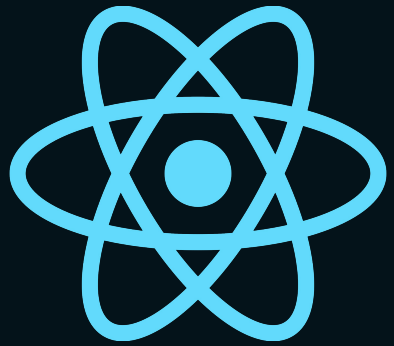
- Podemos também aplicar uma lógica para **adicionar classes a um elemento**;
- Utilizamos o **if ternário**;
- Essa abordagem é mais interessante que o CSS inline, pois o conteúdo da classe está no arquivo de CSS;
- Vamos ver na prática!



CSS Modules

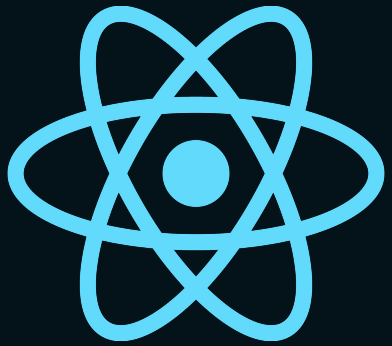
- CSS Modules permite deixar o CSS **scoped**;
- Ou seja, ele só funciona para o componente em questão;
- O nome do arquivo fica: **Component.module.css**;
- É necessário fazer importação também;
- Vamos ver na prática!





CSS no React.js

Conclusão da seção

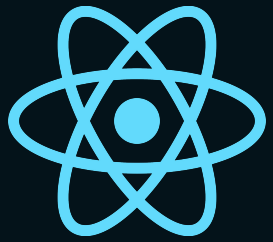


Formulários e React

Introdução da seção

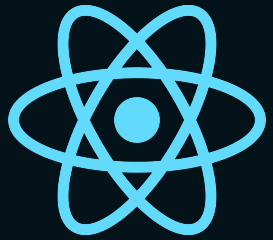
Formulários e React

- Para envio de dados é necessário um formulário, em React também utilizamos a **tag form**;
- As labels tem o atributo for alterado para **htmlFor**, que conta com o name do input;
- **Não utilizamos action**, o envio deve ser feito pelo JavaScript, de forma assíncrona;
- Vamos ver na prática!



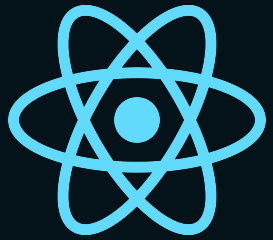
Label envolvendo input

- Em React um padrão muito utilizado é a label ser o elemento **pai do input;**
- O atributo for é opcional nesta abordagem;
- Simplifica o HTML e permanece a **semântica**;
- Vamos ver na prática!



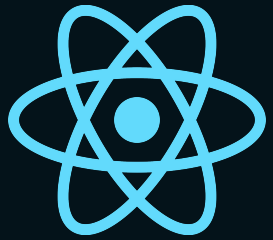
Manipulando valores

- Para manipular os valores de um formulários vamos utilizar o hook **useState**;
- Ou seja, armazenamos o valor com o **set**;
- O evento que vai nos inputs é o **onChange**, e nele teremos a função de alteração;
- Vamos ver na prática!



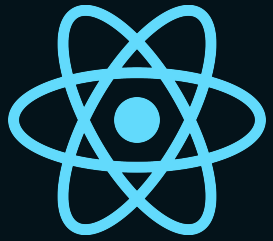
Simplificando a manipulação

- Quando temos diversos inputs no form, podemos simplificar a manipulação;
- Criamos uma **função inline dentro do onChange** e trocamos o valor do dado;
- Vamos ver na prática!



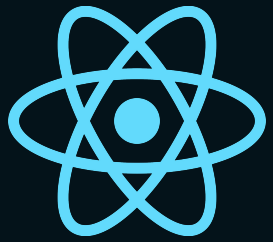
Envio de formulário

- Para enviar formulários utilizamos o evento **onSubmit**;
- Podemos executar uma função, assim como nos inputs;
- Temos que parar o envio do formulário com o **preventDefault**;
- E nesta função é que fazemos **validações** de dados;
- Vamos ver na prática!



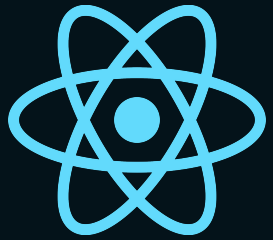
Controlled inputs

- Com o **Controlled input** podemos atribuir valores pré-existentes aos inputs dos forms;
- Precisamos igualar o atributo **value** ao state;
- E também fazer uma lógica que entrega uma **string vazia**, se não houver valor;
- Vamos ver na prática!



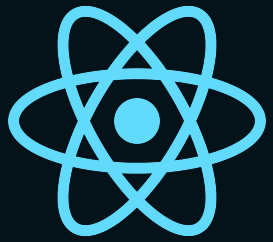
Limpando formulários

- Com o controlled inputs limpar o form fica simples;
- Basta atribuir **valores vazios aos states**;
- Geralmente isso é feito após o envio, para restar o formulário;
- Vamos ver na prática!



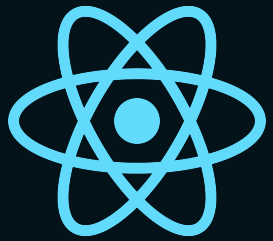
Input de Textarea

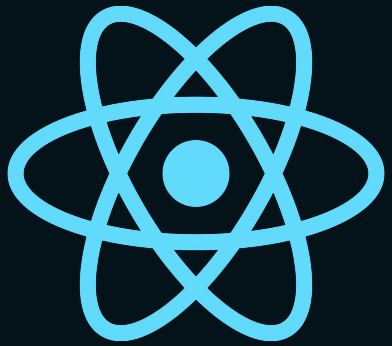
- O **textarea** pode ser aplicado como um **input normal** de texto;
- O atributo **value** pode ser utilizado para mudar o texto inicial;
- E com **onChange** mudamos o seu state;
- Vamos ver na prática!



Input de Select

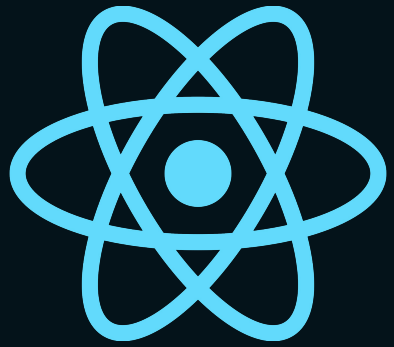
- O **select** é muito semelhante aos outros inputs;
- O evento **onChange** pode mudar o valor do seu state;
- E o **value** deve ser atribuído a uma das options;
- Vamos ver na prática!





Formulários e React

Conclusão da seção

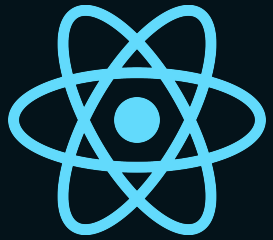


Requisições HTTP e React

Introdução da seção

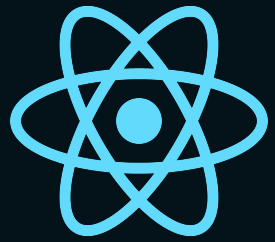
JSON server

- O **JSON server** é um pacote npm;
- Ele **simula uma API**, ou seja, podemos fazer requisições HTTP;
- Vamos integrar este pacote ao React;
- Este é o treino que faremos para APIs reais, que construíremos ao longo dos cursos;
- Isso facilita nossos estudos por **não precisar de um back-end**;
- Vamos criar um projeto e instalar o JSON server!



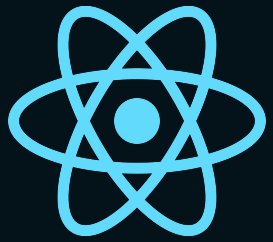
A importância do useEffect

- O **useEffect** faz com que seja possível controlar a execução de uma ação;
- Isso é interessante pois se não o utilizamos recursos podem ser re-executados a cada re-renderização;
- O componente é **re-renderizado a cada mudança**;
- O useEffect possui um **array de dependências** que coordena o que permite a execução do código;
- O useEffect é **muito comum** nas requisições HTTP;



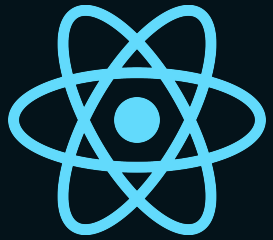
Resgatando dados com React

- Para resgatar dados de uma API temos um procedimento no React;
- Usar **useState** para salvar dados;
- Utilizar **useEffect** para chamar a API apenas quando necessário;
- Realizar a requisição da API com alguma ferramenta, **Axios ou Fetch API**;
- Vamos ver na prática!



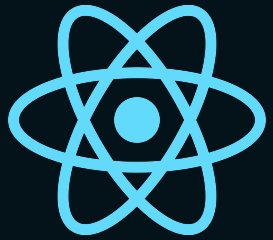
Adicionando dados

- Para adicionar dados via API vamos precisar dos inputs preenchendo os **useStates**;
- Reunimos os dados em uma função, que é disparada no evento de **onSubmit**;
- O verbo HTTP que utilizaremos é o **POST**;
- O processo é parecido com o resgate de dados;
- Vamos ver na prática!



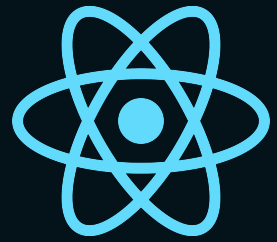
Adicionando dados

- Para adicionar dados via API vamos precisar dos inputs preenchendo os **useStates**;
- Reunimos os dados em uma função, que é disparada no evento de **onSubmit**;
- O verbo HTTP que utilizaremos é o **POST**;
- O processo é parecido com o resgate de dados;
- Vamos ver na prática!



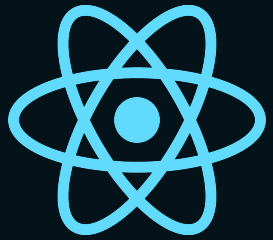
Carregamento dinâmico de dados

- Se a requisição obter êxito, **podemos adicionar no front um novo item a lista**;
- Já temos a informação dele e **não precisamos fazer outra requisição HTTP**;
- Isso deixa nossos projeto mais performático;
- Vamos ver na prática!



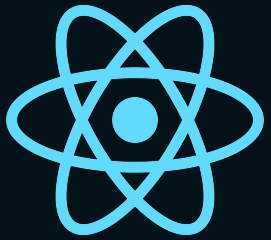
Custom hook para o fetch

- É normal separar as responsabilidades nos componentes;
- Ou seja, termos a função de requisição entre **outro arquivo**;
- Podemos criar o nosso próprio hook para isso;
- Isso é chamado de **custom hook**;
- A pasta geralmente utilizada é a **hooks**;
- Vamos ver na prática!



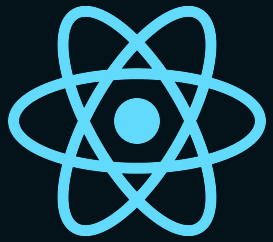
Refatorando o POST

- É possível reutilizar o hook para fazer o **POST**;
- **Vamos criar um useEffect** que mapeia uma outra mudança de estado;
- Após ela ocorrer, adicionamos o produto;
- **Nem sempre reutilizar um hook** para várias ações é a melhor estratégia;
- Vamos ver na prática!



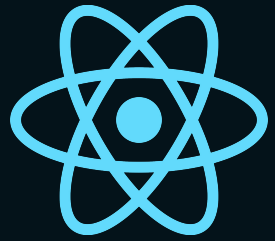
Estado de loading

- Quando fazemos requisições HTTP é normal que a resposta **demora um pouco a chegar;**
- Neste intervalo inserimos um **elemento de loading**;
- É possível inserir no nosso hook esta abordagem;
- Vamos ver na prática!



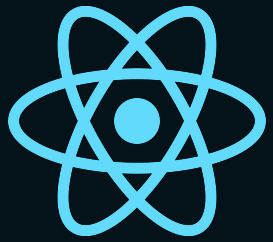
Estado de loading no POST

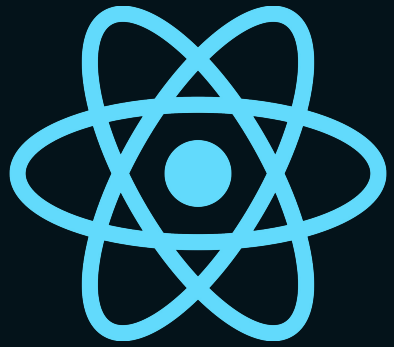
- Podemos bloquear ações **enquanto a requisição ocorre**;
- Isso é interessante para evitar **duplicação de eventos**;
- Podemos identificar um POST ocorrendo, e bloquear o input de envio;
- Vamos ver na prática!



Tratando erros

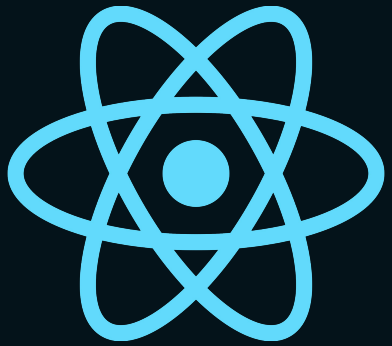
- Podemos tratar erros das requisições com **blocos try catch**;
- **É possível pegar os dados do erro**, e utilizar a mensagem para exibir algo na tela;
- Desta maneira é possível prever erros em todos os cenários do nosso app (resgate, envio, erro);
- Vamos ver na prática!





Requisições HTTP e React

Conclusão da seção

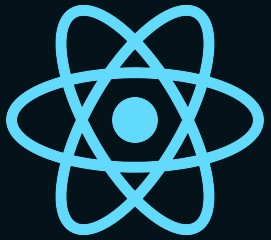


React Router

Introdução da seção

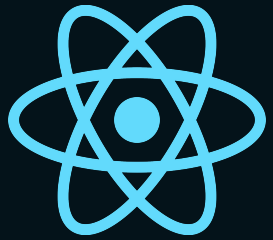
O que é React Router?

- **React Router** é o pacote mais utilizado para criar rotas em uma aplicação React;
- **Cada rota é uma página**, rota é a nomenclatura utilizada;
- Ou seja, permite nosso app SPA ter múltiplas páginas;
- Precisamos instalar e configurar no projeto;
- Há diversas funcionalidades no React Router: **redirecionamento**, **rotas aninhadas**, **404** e etc;
- Vamos criar o projeto e instalá-lo!



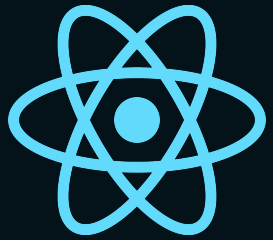
Configurando o React Router

- Para **configurar o React Router** vamos utilizar principalmente o arquivo **main.jsx**;
- Precisamos importar os componentes: **createBrowserRouter**, **RouterProvider**, **Route**;
- Eles serão utilizados na configuração e ao longo do projeto;
- Vamos ver na prática!



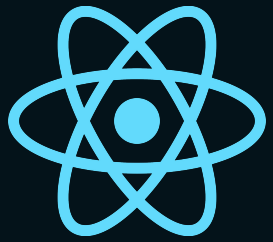
Página de erro / 404

- Podemos criar facilmente uma **página de erro**;
- Precisamos criar um componente, que será a página, geralmente o nome é ErrorPage;
- Depois vamos utilizar o hook **useRouteError** para obter as informações do erro;
- Por último configurar a propriedade **errorElement** em main.jsx como o componente criado;
- Vamos ver na prática!



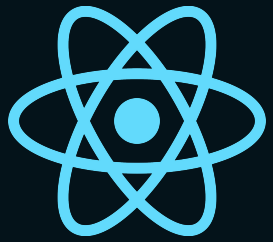
Criando e configurando páginas

- Primeiramente vamos criar o **componente da página**;
- Depois basta inserir um novo objeto em main.jsx.
- Ele deve conter:
  - **path**: caminho para acessar a página;
  - **element**: componente;
- Vamos ver na prática!



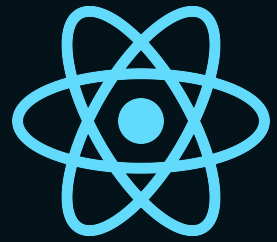
Criando componente base

- O **Outlet** é um componente que nos permite reaproveitar a estrutura das páginas;
- Podemos definir que um componente base seja esta **estrutura**;
- E todas páginas ficam dentro dele;
- As configurações de páginas devem ser feitas na propriedade **children** em main.jsx;
- Vamos ver na prática!



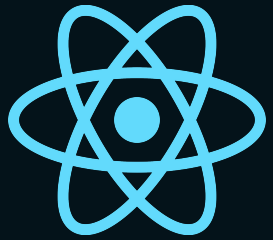
Criando links entre páginas

- Para criar links vamos utilizar o componente **Link**;
- Ele é configurado com a propriedade **to**, que leva a URL de destino;
- Isso permite uma mudança de páginas sem reload;
- Vamos ver na prática!



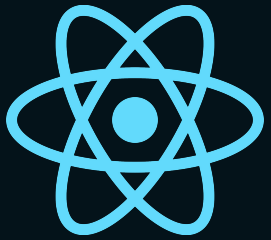
Carregando dados

- Nesta aula vamos utilizar nosso hook **useFetch** para exibir os produtos na Home;
- Isso nos dará possibilidade de explorar **outros recursos do React Router**;
- E também a rever os conceitos aprendidos em requisições de HTTP;
- Vamos ver na prática!



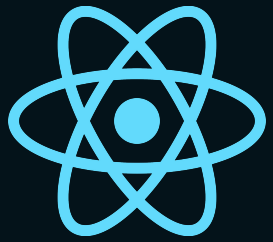
Carregando dados individuais

- O recurso de carregar rotas individuais é chamado de **rota dinâmica**;
- Ou seja, como temos vários produtos a URL de cada um vai variar, dependendo de alguma característica, que geralmente é o id;
- O formato de path é: **/produt/:id**
- Vamos ver na prática!



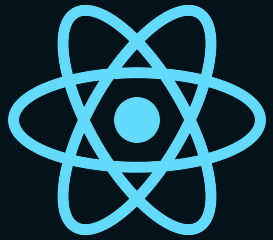
Rotas aninhadas

- As rotas aninhadas ou **nested routes**, são estruturas mais complexas;
- Onde combinamos rotas dinâmicas e criamos uma estrutura maior para acessar a página;
- Exemplo: **/products/:id/info**
- Vamos ver na prática!



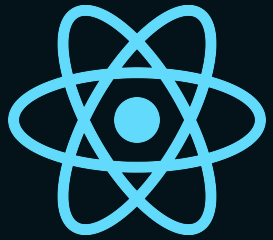
Link ativo

- Para identificar links ativos utilizamos o componente **NavLink** em vez de Link;
- Há uma propriedade isActive que pode **aplicar estilos diferenciados para este link**;
- Vamos ver na prática!



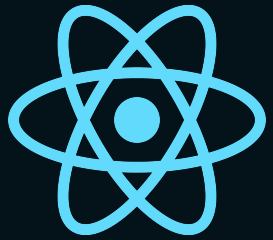
Search Params

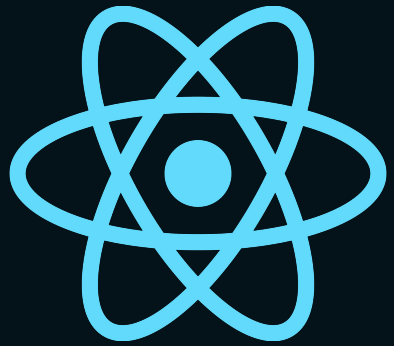
- O recurso de Search Params nos permite pegar **informações da URL**;
- Ele é muito interessante para fazer funcionalidades de busca em um site;
- O hook utilizado para resgatar estes dados é o **useSearchParams**;
- Vamos ver na prática!



Redirect

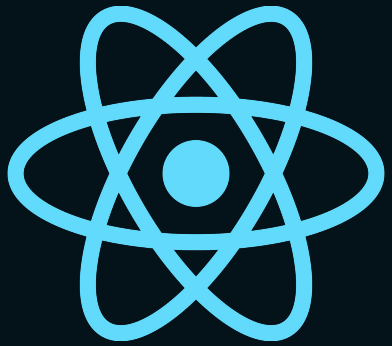
- Podemos criar um redirecionamento de páginas com o componente **Navigate**;
- Exemplo: uma URL que deixa de existir, mas queremos redirecionar o usuário para outra;
- A configuração é feita no próprio **main.jsx**;
- Vamos ver na prática!





React Router

Conclusão da seção

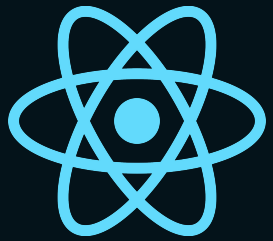


Context API

Introdução da seção

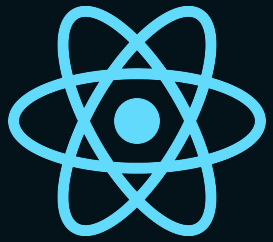
O que é Context API?

- Um recurso do React que facilita o **compartilhamento de dados entre os componentes**;
- Quando há a necessidade de **dados globais**, provavelmente utilizaremos o Context;
- Quando há muitas idas e vindas de props, também deve se considerar o Context;
- Geralmente ficam na **pasta context**;
- Vamos criar um projeto para esta seção!



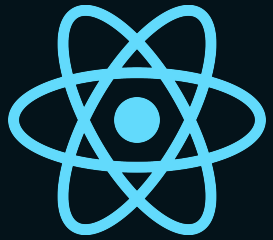
Criando o contexto

- O primeiro passo é **criar o Context**;
- O arquivo tem a **primeira letra maiúscula**, como nos componentes, e geralmente **termina com Context**;
- Exemplo: AlgumContext.js;
- A convenção é deixar na **pasta context** em src;
- Onde utilizamos o contexto, o arquivo precisa ser importado;
- Vamos ver na prática!



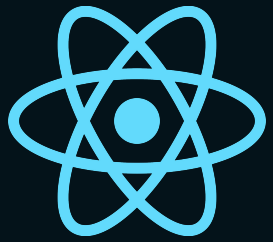
Criando o provider

- O provider vai **delimitar o escopo do contexto**;
- Ou seja, em que componentes teremos acesso aos dados;
- O provider deve **encapsular os componentes** que precisam do context;
- Geralmente é colocado em **main.jsx**;
- O provider tem a **prop children**, para inserirmos elementos dentro;
- Vamos ver na prática!



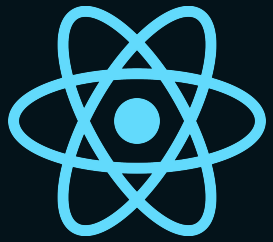
Criando uma estrutura

- Vamos agora criar uma **estrutura mínima** para ver o poder total de Context;
- Iremos instalar o react-router-dom;
- Criar duas páginas;
- E um componente de barra de navegação;
- Vamos lá!



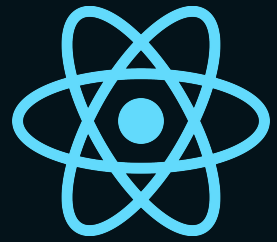
Alterando o contexto

- Agora vamos **exibir e alterar** o valor do contexto;
- Vamos utilizar o hook **useContext** para trazer o nosso contexto as componentes;
- E com este mesmo hook é possível trazer a função que altera o seu valor;
- Vamos ver na prática!



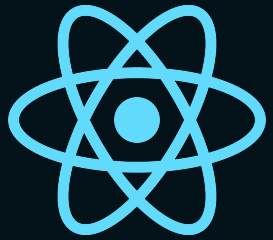
Refatorando contexto com hook

- Podemos **criar um hook** e trabalhar o contexto nele;
- Concentramos o **useContext em um só local**, que será no hook;
- E há um intervalo para uma possível validação na alteração do contexto;
- Vamos ver na prática!



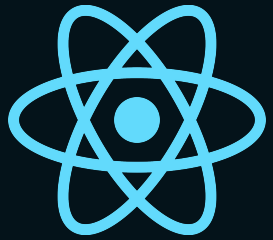
Contexto mais complexo

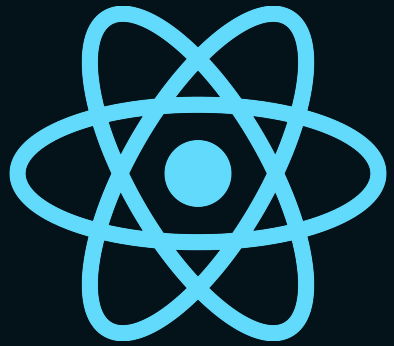
- **Contextos mais complexos** podem necessitar de variações no comportamento;
- Por isso o mais indicado é utilizar o hook **useReducer**;
- Ele funciona como um **useState**, mas com mais possibilidades;
- Vamos ver na prática!



Alterando Contexto complexo

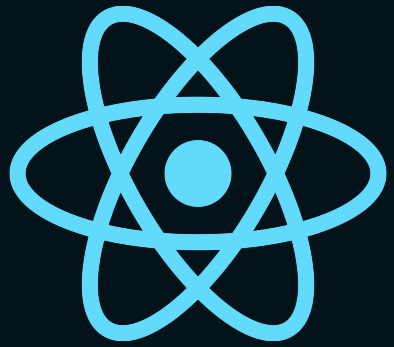
- Para alterar o contexto vamos utilizar uma função chamada **dispatch**;
- Ela também estará no **useReducer**;
- Precisamos enviar todas as informações necessárias para a alteração do valor do contexto;
- Ou seja, o **switch entrará em ação**;
- Vamos ver na prática!





Context API

Conclusão da seção

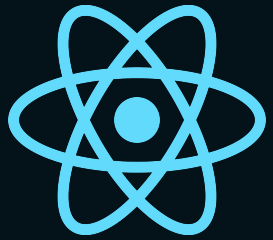


Os hooks do React

Introdução da seção

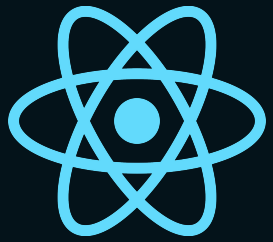
useState

- O **useState** é um dos principais hooks do React;
- Sua função é **gerenciar valores**;
- Podemos consultar e alterar;
- Isso nos permite **re-renderizar um componente**, o que não ocorre na manipulação de variáveis;
- Vamos ver na prática!



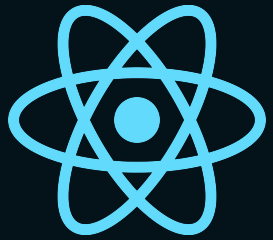
useState e inputs

- Podemos conciliar o **useState** aos valores dos inputs;
- O evento será o **onChange**;
- Com isso podemos salvar valores de um formulário, e posteriormente fazer o envio;
- Há também a estratégia de **Controlled Inputs**;
- Vamos ver na prática!



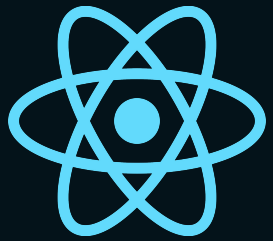
useReducer

- o **useReducer** tem função semelhante ao useState;
- Porém na manipulação de conteúdo podemos **executar uma função**;
- Então temos que o useReducer recebe um valor para gerenciar e uma função para alteração do valor;
- Vamos ver na prática!



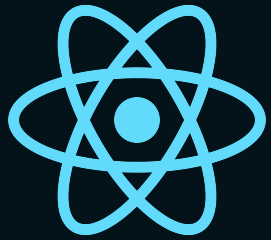
Avançando em useReducer

- Se o **useReducer** fosse utilizado como no exemplo passado, talvez o useState fosse suficiente;
- Por isso o reducer geralmente contém operações mais complexas, utilizando um **switch**;
- Vamos ver na prática!



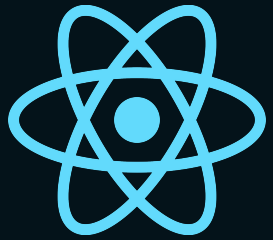
useEffect

- O **useEffect** é utilizado em várias situações, e está no ranking dos mais utilizados no React;
- Podemos fazer **alterações nos elementos** ou **requisições HTTP**;
- A grande vantagem é que **estas ações podem ser controladas**;
- O **array de dependências** possui os itens a serem monitorados;
- Vamos ver na prática!



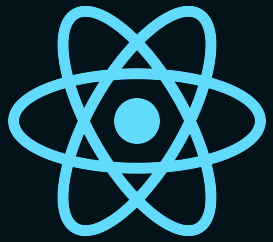
useEffect com array vazio

- As vezes precisamos executar uma ação **uma única vez**;
- Isso pode ser feito com o **array de dependências vazio**;
- Na primeira renderização do componente, o código é executado;
- Vamos ver na prática!



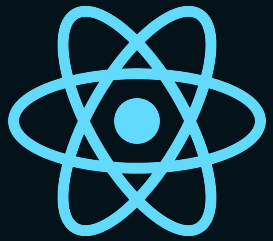
Mais sobre o array de dependências

- Podemos condicionar a execução do useEffect colocando **algo no array de dependências**;
- **Sempre que este dado for modificado**, o useEffect executa mais uma vez;
- Isso permite a **reutilização**, e de forma controlada;
- Vamos ver na prática!



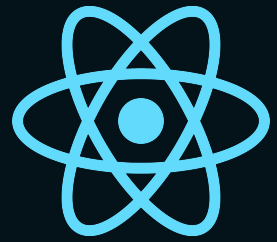
Limpeza do useEffect

- Algumas ações precisam ser limpas com uma técnica chamada de **cleanup**;
- Exemplo: uma ação executada de tempos em tempos, pode ser executada após uma mudança de página;
- Para resolver isso basta finalizar a ação no useEffect com um return;
- Vamos ver na prática!



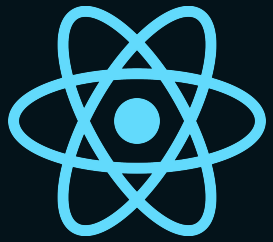
useContext

- O **useContext** é o hook utilizado para consumir um context, da **Context API**;
- Vamos precisar criar o contexto e o provedor (**Provider**);
- Envolver os componentes que vão receber os dados;
- E fazer o uso do hook onde necessário;
- Vamos ver na prática!



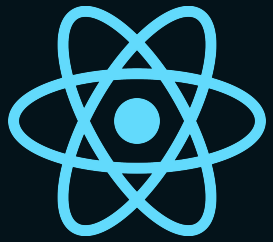
useRef

- O **useRef** pode ser utilizado como o useState, gerenciando valores;
- A diferença é que **ele é um objeto**, e seu valor está na propriedade **current**;
- Este hook **não re-renderiza o componente** ao ser utilizado, o que pode ser interessante em alguns casos;
- Vamos ver na prática!



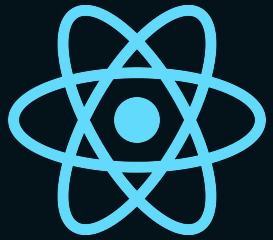
useRef e o DOM

- o useRef pode ser utilizado para **selecionar elementos no JSX**;
- Com isso podemos fazer **manipulação no DOM**, ou aplicar eventos como o **focus**;
- Que deixa o input como selecionado;
- Vamos ver na prática!



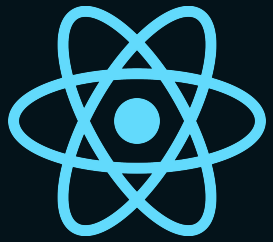
useCallback

- O **useCallback** pode ser utilizado em duas situações;
- Ele basicamente memoriza uma função, fazendo com que ela **NÃO seja reconstruída** a cada renderização;
- O primeiro caos é quando estamos prezando pela performance, então uma **função muito complexa** pode ser criada uma só vez;
- Já o segundo caso é **quando o React nos alerta** que uma função deveria estar no useCallback;
- Vamos ver na prática!



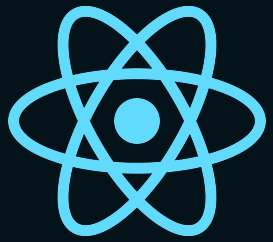
useMemo

- O **useMemo** pode ser utilizado para garantir a referência de um objeto;
- Fazendo com que algo possa ser atrelado a uma referência e não um valor;
- Com isso é possível **condicionar useEffects a uma variável** de maneira mais inteligente;
- Vamos ver na prática!



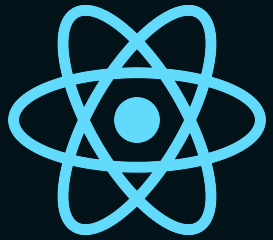
useLayoutEffect

- Muito parecido com o **useEffect**;
- A grande diferença é que este **hook roda antes da renderização** do componente;
- Ou seja, **o hook é síncrono**, bloqueando o carregamento da página para o sucesso da funcionalidade;
- A ideia é executar algo antes de qualquer elemento aparecer na página;
- Vamos ver na prática!



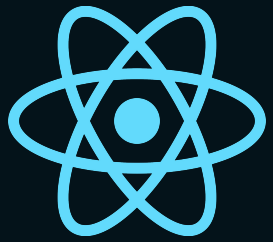
useImperativeHandle

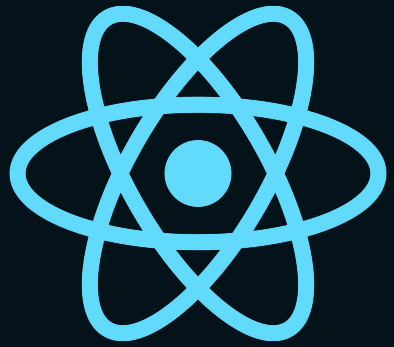
- Com o **useImperativeHandle** temos como acionar ações em outro componente, de forma imperativa;
- Como não podemos passar refs como props, precisamos utilizar a função **fowardRef**;
- Isso nos permite passar referências e torna nosso exemplo viável;
- Vamos ver na prática!



Custom hooks

- Os **Custom hooks** são os hooks que nós criamos;
- Estratégia utilizada para **abstrair funções complexas** do componente ou **reaproveitar** o código;
- Muito utilizado em projetos profissionais;
- Vamos ver na prática!





Os hooks do React

Conclusão da seção

Curso de TypeScript

Uma introdução ao superset

O que é TypeScript?

- TypeScript é um **superset** para a linguagem JavaScript;
- Ou seja, adiciona funções ao JavaScript, como a **declaração de tipos** de variável;
- Pode ser utilizado com frameworks/libs, como: **Express e React**;
- Precisa ser **compilado em JavaScript**, ou seja, não executamos TS;
- Desenvolvido e mantido pela **Microsoft**;

Por que TypeScript?

- Adiciona **confiabilidade** ao programa (tipos);
- Provê novas funcionalidades a JS, como **Interfaces**;
- Com TS podemos **verificar os erros antes da execução** do código, ou seja, no desenvolvimento;
- Deixa JavaScript **mais explícito**, diminuindo a quantidade de bugs;
- Por estes e outros motivos **perdemos menos tempo com debug**;

Instalando o TypeScript

- Para instalar o TypeScript vamos utilizar o **npm**;
- O nome do pacote é **typescript**;
- E vamos adicionar de forma global com a **flag -g**;
- A partir da instalação temos como **executar/compilar** TS em qualquer local da nossa máquina, com o comando **tsc**;

Primeiro programa

- Vamos criar nosso **primeiro programa em TS**;
- O intuito é entender como **compilar e executar** o arquivo gerado pelo processo de compilação;
- Vamos a prática!

Live Server

- A cada vez que geramos o arquivo **temos que recarregar a página**, o que pode se tornar chato ao longo tempo;
- A extensão **LiveServer** faz com que a cada alteração de código, o navegador atualize automaticamente;
- **Inclusive o compile** provoca a atualização;
- Vamos instalar!

Como tirar o melhor proveito do curso

- **Programar junto**, e não ficar só assistindo;
- Criar seus **próprios exemplos** (para os conteúdos das aulas);
- **Projetos próprios** ao final do curso ou durante;
- Realizar os **exercícios** propostos;
- **Responder** ou criar novas perguntas no Q&A;
- Dica extra: ver depois fazer;

Introdução ao TS

Conclusão

Desafio 1

1. Vamos criar um novo projeto para treinar a compilação e execução de código;
2. O desafio é criar uma função que aceita dois argumentos, somente do tipo numérico (number);
3. E exibe a soma entre os dois;

Fundamentos do TS

Introdução

O que são tipos?

- Em TypeScript a principal função é **determinar tipos para os dados**;
- Isso vai garantir a **qualidade do código**;
- Além de fazer o TS nos **ajudar na hora do desenvolvimento**;
- Ou seja, precisamos **definir corretamente o tipo** das variáveis, dos retornos das funções, das manipulações de dados;
- Consequentemente teremos um software melhor programado e é este o principal intuito do TS;

Tipos primitivos

- Há diversos tipos em TS, vamos começar pelos **primitivos**;
- Que são: **number**, **string** e **boolean**;
- Todos estes são inseridos com **letras minúsculas**;
- E por mais óbvio que pareça, eles servem para: números, textos e booleanos;

Conhecendo o number

- O tipo **number** garante que o dado seja um número;
- Logo, podemos **inserir apenas números** na variável;
- E também mudar o valor para outro número;
- O TypeScript **possibilita também a inserção de métodos numéricos** apenas;
- Vamos ver na prática;

Conhecendo o string

- O tipo **string** garante que o dado seja um texto;
- Logo, podemos **inserir apenas texto** na variável;
- E também mudar o valor para outro texto;
- O TypeScript **possibilita também a inserção de métodos de texto** apenas;
- Vamos ver na prática;

Conhecendo o boolean

- O tipo **boolean** garante que o dado seja um booleano (true ou false);
- Logo, podemos **inserir apenas booleans** na variável;
- E também mudar o valor para outro boolean;
- Vamos ver na prática;

TS e a aplicação

- Talvez já tenha ficado claro, mas programar com TS é como um **pair programming**;
- Temos sempre alguém para nos avisar se algo é **feito errado**;
- Depois da compilação **o TS não tem mais efeito**, ele não pode mais nos ajudar;
- Por isso há **uma trava de compilação com erros**;
- Além de erros, o TS também proporciona **avisos**;

Type annotation e Type inference

- Estes dois conceitos vão nos acompanhar em **todo o processo do desenvolvimento** de aplicações;
- **Annotation** é quando definimos o tipo de um dado manualmente;
- **Inference** é quando o TS identifica e define o tipo de dados para nós;
- Futuramente entraremos em mais alguns detalhes sobre estes recursos;
- Vamos ver na prática!

Gerando arquivo de configuração

- O TS pode ser configurado de **muitas maneiras**;
- Mas para isso precisamos do **arquivo de configuração**;
- Para criar ele utilizamos: **tsc --init**
- Agora podemos mudar várias opções em relação ao que o TS executa e também feito o compile;

Compilar TS automaticamente

- Estamos **sofrendo** muito, não é?
- Para gerar a compilação automática podemos utilizar o comando: **tsc -w**
- O **nosso output será gerado automaticamente sempre que salvarmos** o projeto;
- Agora vai!

Fundamentos do TS

Conclusão

Desafio 2

1. Crie uma variável que recebe um número;
2. Converta este número para string em uma nova variável;
3. Esta variável de conversão deve estar tipada por inferência;
4. Imprima este número em uma string maior, utilizando o recurso de template strings ou concatenação;

Avançando em tipos

Introdução

Arrays

- Podemos especificar um **array** como tipo também;
- Se temos um array de números: **number[]**
- Se é um array de string: **string[]**
- Isso acontece pois **geralmente os arrays possuem apenas um único tipo** de dado entre seus itens;
- Vamos ver na prática!

Outra sintaxe de arrays

- Os tipos de array possuem duas sintaxes;
- **Obs:** a da primeira aula é a mais utilizada;
- Podemos também fazer desta maneira: **Array<number>**
- Vamos ver na prática!

O tipo any

- O **any** transmite ao TS que qualquer tipo satisfaz a variável;
- Devemos **evitar ao máximo evitar este tipo**, pois vai contra os princípios do JavaScript;
- **Dois casos de uso:** o tipo do dado realmente não importa e arrays com dados de múltiplos tipos;
- Vamos ver na prática!

Tipo de parâmetro de funções

- Podemos **definir o tipo de cada parâmetro de uma função**;
- Assim condicionamos o seu uso correto;
- A sintaxe é: function minhaFuncao(**nome: string**) { }
- Agora podemos passar o parâmetro nome, **apenas como texto**;
- Vamos ver na prática!

Tipo de retorno de funções

- Os **retornos** também podem ser definidos por nós;
- Para isso utilizamos a sintaxe: function myFunction()**: number { }**
- Esta função **retorna um número**;
- Vamos ver na prática!

Funções anônimas em TS

- O TypeScript consegue nos **ajudar também em funções anônimas**;
- **Fazendo uma validação do código digitado**, nos fornecendo dicas de possíveis problemas;
- Exemplo: métodos não existentes;
- Vamos ver na prática!

Tipos de objetos

- Podemos determinar tipos para objetos também;
- A sintaxe é: **{prop: tipo, prop2: tipo2}**
- Ou seja, estamos determinando quais tipos as **propriedades de um objeto possuem**;
- Vamos ver na prática!

Propriedades opcionais

- Nem sempre os objetos possuem todas as propriedades que poderiam possuir;
- Por isso temos as **propriedades opcionais**;
- Para ter esse resultado devemos colocar uma interrogação: **{nome: string, sobrenome?: string}**
- Vamos ver na prática!

Validação de props opcionais

- Quando a propriedade é opcional, **precisamos criar uma validação**;
- Isso acontece **por que o TypeScript não nos ajuda mais**, já que ele deixa de controlar o valor que recebemos;
- Para isto utilizamos uma **condicional if**, e conseguimos resolver a situação;
- Vamos ver na prática!

Union types

- O **Union type** é uma alternativa melhor do que o any;
- Onde podemos **determinar dois tipos** para um dado;
- A sintaxe: **number | string**
- Vamos ver na prática!

Avançando com Union Types

- Podemos utilizar **condicionais** para validação do tipo de union types;
- Com isso é possível **trilhar rumos diferentes**, baseado no tipo de dado;
- Para checar o tipo utilizamos **typeof**;
- Vamos ver na prática!

Type alias

- **Type alias** é um recurso que permite criar um tipo e determinar o que ele verifica;
- Desta maneira **temos uma maneira mais fácil de chamá-lo** em vez de criar expressões complexas com Union types, por exemplo;
- Vamos ver na prática!

Introdução às interfaces

- Uma outra maneira de **nomear um tipo de objeto**;
- Podemos **determinar um nome** para o tipo;
- E também escolher **quais as propriedades e seus tipos**;
- Vamos ver na prática!

Diferença entre type alias e interfaces

- Na maioria das vezes **podemos optar entre qualquer um dos recursos** sem problemas;
- A única diferença é que **a Interface pode ser alterada ao longo do código**, já o alias não;
- Ou seja, se pretendemos mudar como o tipo se conforma, devemos escolher a Interface;
- Vamos ver na prática!

Literal types

- **Literal types** é um recurso que permite colocar valores como tipos;
- Isso restringe o uso a não só tipos, **como também os próprios valores**;
- Este recurso é **muito utilizado com Union types**;
- Vamos ver na prática!

Non-null Assertion Operator

- Às vezes o TypeScript pode evidenciar um erro, **baseado em um valor que no momento do código ainda não está disponível**;
- Porém se sabemos que este valor será preenchido, podemos evitar o erro;
- Utilizamos o caractere **!**
- Vamos ver na prática!

Bigint

- Com o tipo **bigint** podemos declarar números com valores muito altos;
- Podemos utilizar a **notação literal**, exemplo: 100n
- Para este recurso precisamos **mudar a configuração do TS**, para versão mínima de ES2020
- Vamos ver na prática!

Symbol

- De forma resumida, o **Symbol** cria uma referência única para um valor;
- Ou seja, mesmo que ele possua o mesmo valor de outra variável, **teremos valores sendo considerados diferentes**;
- Vamos ver na prática!

Avançando em tipos

Conclusão

Narrowing

Introdução

O que é narrowing?

- **Narrowing** é um recurso de TS para identificar tipos de dados;
- Dando assim uma direção diferente a execução do programa, **baseada no tipo que foi identificado**;
- Há situações em que **os tipos podem ser imprevisíveis**, e queremos executar algo para cada uma das possibilidades;
- Isso é fundamental também para **evitar erros do compilador**, identificando e resolvendo os possíveis erros na hora do desenvolvimento;

Typeof type guard

- O **type guard** é basicamente uma validação do tipo utilizando o typeof;
- Desta maneira podemos **comparar o retorno do operador** com um possível tipo;
- **Todos os dados vem como string**, exemplo: “string”, “number”, “boolean”
- E a partir disso realizamos as bifurcações;
- Vamos ver na prática!

Checando se valor existe

- Em JavaScript podemos **colocar uma variável em um if**, e se houver algum valor recebemos um true;
- Quando não há valor um false;
- **Desta maneira conseguimos realizar o narrowing também**, é uma outra estratégia bem utilizada;
- Vamos ver na prática!

Operador instanceof

- Para além dos tipos primitivos, podemos trabalhar com **instanceof**;
- Checando se um dado pertence a uma determinada classe;
- E ele vai servir até para as **nossas próprias classes**;
- Vamos ver na prática!

Operador in

- O operador in é utilizado para **checar se existe uma propriedade no objeto**;
- Outro recurso interessante para o narrowing;
- Pois propriedades também podem ser **opcionais**;
- Vamos ver na prática!

Narrowing

Conclusão da seção

Desafio 3

1. Vamos criar uma função que recebe review dos usuários, precisamos utilizar o narrowing para receber o dado;
2. As possibilidades são boolean e number;
3. Serão enviados números de 1 a 5 (estrelas), prever uma mensagem para cada uma destas notas;
4. Ou false, que é quando o usuário não deixa uma review, prever um retorno para este caso também;

Aprofundando em funções

Introdução da seção

Funções que não retornam nada

- Podemos ter funções que não retornam valores;
- Qual seria o **tipo de dado** indicado para essa situação?
- Neste caso utilizamos o **void**!
- Ele vai dizer ao TS que **não temos um valor de retorno**;
- Vamos ver na prática!

Callback como argumento

- Podemos inserir uma **função de callback** como argumento de função;
- Nela conseguimos **definir o tipo de argumento aceito pela callback**;
- E também o **tipo de retorno** da mesma;
- Vamos ver na prática!

Generic functions

- É uma estratégia para quando **o tipo de retorno é relacionado ao tipo do argumento**;
- Por exemplo: um item de um array, pode ser string, boolean ou number;
- Normalmente são utilizadas **letras como T ou U** para definir os generics, é uma convenção;
- Vamos ver na prática!

Constraints nas Generic Functions

- As Generic Functions podem ter seu **escopo reduzido por constraints**;
- Basicamente **limitamos os tipos que podem ser utilizados** no Generic;
- Isso faz com que nosso escopo seja menos abrangente;
- Vamos ver na prática!

Definindo tipo de parâmetros

- Em Generic functions **temos que utilizar parâmetros de tipos semelhantes**, se não recebemos um erro;
- Porém há a possibilidade de **determinar o tipo também dos parâmetros aceitos**, com uma sintaxe especial;
- Isso faz com que a validação do TS aceite os tipos escolhidos;
- Vamos ver na prática!

Parâmetros opcionais

- Nem sempre utilizamos **todos os parâmetros** de uma função;
- Mas se ele for opcional, precisamos **declarar isso para o TS**;
- E ainda deixar ele no **fim da lista** de parâmetros;
- Vamos ver na prática!

Parâmetros default

- Os **parâmetros default** são os que já possuem um valor definido;
- Se não passarmos para a função, é utilizado esse valor;
- Para este recurso, **a aplicação em TS é igual JS**;
- Vamos ver na prática!

O tipo unknown

- O **tipo unknown** é utilizado de forma semelhante ao any, ele aceita qualquer tipo de dado;
- Porém a diferença é que ele **não deixa algo ser executado** se não houver validação de tipo;
- Ou seja, adiciona uma **trava de segurança**;
- Vamos ver na prática!

O tipo never

- O **never** é um tipo de retorno semelhante ao void;
- Porém é utilizado quando a função **não retorna nada**;
- Um exemplo: retorno de erros;
- Vamos ver na prática!

Rest parameters

- Em JavaScript ES6 podemos utilizar o **Rest Operator**;
- Para aplicá-lo em parâmetros em TS é fácil, basta **definir o tipo de dado com a sintaxe de Rest** (...);
- Vamos ver na prática!

Destructuring em parâmetros

- O **Destructuring**, outro recurso de ES6, também pode ser aplicado com TS;
- Precisamos apenas **determinar o tipo de cada dado que será desestruturado**;
- Desta maneira o TS valida o Destructuring;
- Vamos ver na prática!

Aprofundando em funções

Conclusão da seção

Object Types

Introdução da seção

O que são Object Types?

- São os dados que tem como o tipo objeto, por exemplo: **object literals** e **arrays**;
- Temos **diversos recursos** para explorar sobre estes tipos;
- Como: **Interfaces**, **readonly**, **tupla** e outros;
- Isso nos dá possibilidades a mais para o JavaScript;
- Nesta seção focaremos nestes detalhes que são **super importantes para o TypeScript;**

De tipo para Interface

- Um caso de uso para interfaces é **simplificar os parâmetros de objeto** de funções;
- Ou seja, em vez de passar parâmetros de um objeto longo para n funções, **passamos apenas a interface**;
- Vamos ver na prática!

Propriedades opcionais em interfaces

- As interfaces podem conter **propriedades de objeto opcionais**;
- Basta adicionar a **interrogação** no nome da propriedade;
- Exemplo: **nome?: string**
- Vamos ver na prática!

Propriedades readonly

- As propriedades **readonly** podem ser alteradas apenas uma vez, na criação do novo dado;
- É uma forma de criar um **valor constante** em um objeto;
- Podemos adicionar as **interfaces**;
- Vamos ver na prática!

Index Signature

- Utilizamos o **Index Signature** quando não sabemos o nome das chaves, **mas já sabemos quais os tipos de um objeto ou array**;
- Isso **restringe** a tipos que não devem ser utilizados;
- Uma barreira de segurança a mais na nossa variável;
- Vamos ver na prática!

Extending Types

- Utilizamos **Extending Types** como uma herança para criar tipos mais complexos por meio de uma interface;
- Ou seja, uma interface pode **herdar as propriedades e tipos já definidos** de outra;
- Isso acontece por meio da instrução **extends**;
- Vamos ver na prática!

Intersection Types

- **Intersection Types** são utilizados para criar tipos mais complexos a partir de duas interfaces, por exemplo;
- Podemos concatenar os tipos com **&**;
- Vamos ver na prática!

Extending x Intersection

- Qual devemos utilizar?
- Os dois recursos **chegam no mesmo ponto**, implementando tipos mais complexos;
- Ou seja, **vai da sua escolha** optar por um deles;
- Minha opinião: o Extending tem uma **sintaxe parecida com OOP**;
- Além disso a sintaxe é mais enxuta, não precisamos **criar um Type**, por exemplo;

ReadOnlyArray

- O **ReadOnlyArray** é um tipo para arrays, que deixa os itens como readonly;
- Podemos **aplicar um tipo para os itens do array**, além desta propriedade especial;
- **A modificação de itens pode ser feita através de método**, mas não podemos aumentar o array, por exemplo;
- Vamos ver na prática!

Tuplas

- **Tupla é um tipo de array**, porém definimos a quantidade e o tipo de elementos;
- Basicamente **criamos um novo type**, e nele **inserimos um array com os tipos necessários**;
- Cada tipo conta também como um elemento;
- Vamos ver na prática!

Tuplas com readonly

- Podemos criar **tuplas com a propriedade de readonly**;
- É um tipo de dado **super restrito** pois:
- Limita quantos itens teremos, qual o tipo de cada um e também não são modificáveis;
- Vamos ver na prática!

Object Types

Conclusão da seção

Criação de Tipos

Introdução da seção

Sobre a criação de novos tipos

- Há a possibilidade de **gerar novos tipos em TypeScript**, já vimos isso com **Generics** (vamos nos aprofundar neste recurso também);
- Porém existem outros operadores que visam facilitar nossa vida neste assunto;
- A ideia deste recurso é **deixar a manutenção do projeto mais simples**;
- Ou seja, gastar mais tempo na estruturação dos tipos e menos na busca de possíveis bugs depois;

Generics

- Utilizamos **Generics** quando uma função **pode aceitar mais de um tipo**;
- É uma prática **mais interessante do que o any**, que teria um efeito semelhante;
- Porém com Generics
- Vamos ver na prática!

Constraint em Generics

- As constraints nos ajudam a **limitar os tipos aceitos**;
- **Como em Generic podemos ter tipos livres**, elas vão filtrar os tipos aceitos;
- Adicionando organização quando queremos aproveitar a liberdade dos Generics;
- Vamos ver na prática!

Interfaces com Generics

- Com **Interfaces** podemos criar tipos complexos para objetos;
- **Adicionando Generics** podemos deixá-los mais brandos;
- Aceitando tipos diferentes em ocasiões diferentes;
- Vamos ver na prática!

Type parameters

- **Type parameters** é um recurso de Generics;
- Utilizado para dizer que **algum parâmetro de uma função**, por exemplo, **é a chave de um objeto**, que também é parâmetro;
- Desta maneira conseguimos criar uma **ligação entre o tipo genérico e sua chave**
- Vamos ver na prática!

keyof Type Operator

- Com o **keyof Type Operator** podemos criar um novo tipo;
- Ele aceita **dados do tipo objeto**, como object literals e arrays;
- E pode criar o tipo baseado nas **chaves do objeto** passado como parâmetro;
- Vamos ver na prática!

typeof Type Operator

- Com o **typeof Type Operator** podemos criar um novo tipo;
- Este tipo é será **baseado no tipo de algum dado**;
- Ou seja, é interessante para quando queremos criar uma variável com o mesmo tipo da outra, por exemplo;
- Vamos ver na prática!

Indexed Access types

- A abordagem **Indexed Access types** pode criar um tipo baseado em uma chave de objeto;
- Ou seja, conseguimos reaproveitar o tipo da chave para outros locais, como funções;
- Vamos ver na prática!

Conditional Expressions Type

- O **tipo por condição** permite criar um novo tipo com base em um if/else;
- Mas não são aceitas expressões tão amplas;
- Utilizamos a sintaxe de **if ternário**;
- Vamos ver na prática!

Template Literals Type

- Podemos criar tipos com **Template literals**;
- É uma forma de customizar tipos **de maneiras infinitas**;
- Pois o texto que formamos pode depender de variáveis;
- Vamos ver na prática!

Criação de Tipos

Conclusão da seção

Classes

Introdução da seção

Campos em classes

- Em TS podemos **adicionar novos campos a uma classe**, ou seja, iniciar a classe com campos para os futuros dados dos objetos;
- Que serão as **propriedades** dos objetos instanciados;
- Estes campos podem ser **tipados** também;
- Note que um campo sem valor inicial, deve ser declarado com **!**;
- Vamos ver na prática!

Constructor

- **Constructor** é uma função que nos dá a possibilidade de iniciar um objeto com valores nos seus campos;
- Isso faz com que **não precisemos mais da !**;
- Provavelmente **todos os campos serão preenchidos** na hora de instanciar um objeto;
- Vamos ver na prática!

Campos readonly

- Podemos iniciar o campo com valor e torná-lo **readonly**;
- Ou seja, será um **valor só para consulta**;
- Não poderemos alterar este valor ao longo do programa;
- Vamos ver na prática!

Herança e super

- Para gerar uma herança utilizamos a palavra reservada **extends**;
- Depois vamos **precisar passar as propriedades da classe pai para ela**, quando instanciamos um objeto;
- Isso será feito com a **função super**;
- Vamos ver na prática!

Métodos

- **Métodos** são como funções da classe;
- **Podemos criá-los junto das classes** e os objetos podem utilizá-los;
- É uma maneira de **adicionar funcionalidades** as classes;
- Vamos ver na prática!

O this

- A palavra reservada this serve para **nos referirmos ao objeto em si**;
- Ou seja, conseguimos **acessar as suas propriedades**;
- Esta funcionalidade funciona da mesma forma que em JavaScript;
- Vamos ver na prática!

Utilizando getters

- Os **getters** são uma forma de retornar propriedades do objeto;
- Porém **podemos modificá-las neste retorno**, isso é muito interessante;
- Ou seja, é um método para ler propriedades;
- Vamos ver na prática!

Utilizando setters

- Os getters leem propriedades, os **setters** as modificam/atribuem;
- Logo, **podemos fazer validações antes de inserir** uma propriedade;
- Os setters também **funcionam como métodos**;
- Vamos ver na prática!

Herança de interfaces

- Podemos herdar de interfaces também com a instrução **implements**;
- A ideia é bem **parecida de extends**;
- Porém esta forma é mais utilizada para **criar os métodos que várias classes terão em comum**;
- Vamos ver na prática!

Override de métodos

- O **override** é uma técnica utilizada para substituir um método de uma classe que herdamos algo;
- Basta **criar o método com o mesmo nome** na classe filha;
- Isso caracteriza o override;
- Vamos ver na prática!

Visibilidade

- Visibilidade é o conceito de expor nossos métodos de classes;
- **public**: visibilidade default, pode ser acessado em qualquer lugar;
- **protected**: acessível apenas a subclasses da classe do método, para acessar uma propriedade precisamos de um método;
- **private**: apenas a classe que declarou o método pode utilizar;
- Veremos exemplos de todos eles a seguir!

Visibilidade: public

- O **public** é o modo de visibilidade default;
- Ou seja, **já está implícito** e não precisamos declarar;
- Basicamente **qualquer método ou propriedade** de classe pai, estará acessível na classe filha;
- Vamos ver na prática!

Visibilidade: protected

- Os itens **protected** podem ser utilizados apenas em subclasses;
- Uma propriedade **só pode ser acessada por um método**, por exemplo;
- O mesmo acontece com métodos;
- Adicionando uma camada de segurança ao código criado em uma classe;
- Vamos ver na prática!

Visibilidade: private

- Os itens **private**, propriedades e objetos, só podem ser acessados na classe que os definiu;
- E ainda **precisam de métodos** para serem acessados;
- Esta é a **maior proteção** para propriedades e métodos;
- Vamos ver na prática!

Static members

- Podemos criar propriedades e métodos **estáticos** em classes;
- Isso faz com que o acesso ou a utilização **não dependam de objetos**;
- Podemos utilizá-los a partir **da própria classe**;
- Vamos ver na prática!

Generic class

- Podemos criar classes com **tipos genéricos** também;
- Ou seja, as propriedades dos argumentos podem ter os tipos definidos **na hora da criação da instância**;
- Isso nos permite **maior flexibilidade** em uma classe;
- Vamos ver na prática!

Parameters properties

- **Parameters properties** é um recurso para definir a privacidade, nome e tipo das propriedades no constructor;
- Isso **resume um pouco a sintaxe** das nossas classes;
- Vamos ver na prática!

Class Expressions

- **Class Expressions** é um recurso para criar uma classe anônima;
- Podemos também utilizar **generics** junto deste recurso;
- Vamos encapsular a classe em uma **variável**;
- Vamos ver na prática!

Abstract class

- **Abstract Class** é um recurso para servir como molde de outra classe;
- **Todos os métodos dela devem ser implementados** nas classes que a herdam;
- E também **não podemos instanciar objetos** a partir destas classes;
- Vamos ver na prática!

Relações entre classes

- Podemos criar um **objeto com base em outra classe**;
- **Quando as classes são idênticas** o TS não reclama sobre esta ação;
- Precisamos que as duas sejam exatamente iguais;
- Vamos ver na prática!

Classes

Conclusão da seção

Trabalhando com Módulos

Introdução da seção

Introdução aos módulos

- Os módulos são a forma que temos para importar código em arquivos;
- Podemos exportar código com **export default**;
- E importar com **import**;
- **Criaremos os arquivos com .ts**, mas importaremos como .js;
- Isso nos dá mais flexibilidade, podendo separar as responsabilidades em arquivos;
- Utilizaremos o **Node.js** para executar os arquivos com módulos;

Importando arquivos

- Para começar **vamos criar um arquivo simples e importar seu conteúdo**;
- Basta criar um **arquivo .ts**, desenvolver o código e exportar;
- Depois no arquivo principal vamos importar o arquivo anterior, com a **extensão .js**;
- Vamos ver na prática!

Importando variáveis

- Podemos **exportar e importar variáveis** também;
- A sintaxe é um pouco mais simples, vamos utilizar **apenas o export**;
- No arquivo de importação vamos importar os valores com **destructuring**;
- Vamos ver na prática!

Múltiplas importações

- Podemos **exportar múltiplas variáveis e funções**;
- Isso pode ser realizado no **mesmo arquivo**;
- Para esta modalidade utilizamos **export para todos os dados**;
- E todos devem ser importados com destructuring;
- Vamos ver na prática!

Alias para importações

- Podemos criar um **alias** para importações;
- Ou seja, **mudar o nome** do que foi importado;
- Podendo tornar este novo nome, uma **forma mais simples de chamar o recurso**;
- Vamos ver na prática!

Importando tudo

- Podemos **importar tudo que está em um arquivo** com apenas um símbolo;
- Utilizamos o * para isso;
- Os dados virão em um **objeto**;
- Vamos ver na prática!

Importando tipos

- Importar **tipos ou interfaces** também é possível;
- Vamos exportar como se fossem **variáveis**;
- E no arquivo que os recebe, utilizamos **destructuring**;
- Depois podemos implementar no projeto;
- Vamos ver na prática!

Trabalhando com Módulos

Conclusão da seção

Decorators

Introdução da seção

O que são os decorators?

- Decorators podem **adicionar funcionalidades extras** a classes e funções;
- Basicamente criamos novas funções, que são adicionadas a partir de um **@nome**;
- Esta função será chamada assim que o item que foi definido o decorator **for executado**;
- Para habilitar precisamos adicionar uma configuração no **tsconfig.json**;

Primeiro decorator

- Vamos criar um decorator como uma **function**;
- Ele pode trabalhar com argumentos especiais que são: **target**, **propertyKey** e **descriptor**;
- Estes são os **grandes trunfos** do decorator, pois nos dão informação do local em que ele foi executado;
- Vamos ver na prática!

Múltiplos decorators

- Podemos utilizar **múltiplos decorators** em TS;
- O primeiro a ser executado é o que está **mais ao topo do código**;
- Desta maneira é possível criar operações mais complexas;
- Vamos ver na prática!

Decorator de classe

- O decorator de classe está ligado ao **constructor**;
- Ou seja, sempre que este for executado, **teremos a execução do decorator**;
- Isso nos permite acrescentar algo a criação de classes;
- Vamos ver na prática!

Decorator de método

- Com este decorator podemos **modificar a execução de métodos**;
- Precisamos inserir o decorator **antes da declaração do método**;
- Ele é executado antes do método;
- Vamos ver na prática!

Accessor decorator

- Semelhante ao decorator de método;
- Porém este serve apenas para os **getters e setters**;
- Podemos alterar a execução antes de um set ou get;
- Vamos ver na prática!

Property decorator

- O **property decorator** é utilizado nas propriedades de uma classe;
- Ou seja, na hora da definição da mesma podemos **ativar uma função**;
- Isso nos ajuda a modificar ou validar algum valor;
- Vamos ver na prática!

Exemplo real: Class Decorator

- Com **Class Decorator** podemos influenciar o constructor;
- Neste exemplo vamos criar uma função para inserir **data de criação dos objetos**;
- Vamos ver na prática!

Exemplo real: Method Decorator

- Com **Method Decorator** podemos alterar a execução dos métodos;
- Neste exemplo vamos **verificar se um usuário pode ou não fazer uma alteração** no sistema;
- A alteração seria o método a ser executado;
- Vamos ver na prática!

Exemplo real: Property Decorator

- Com o **Property Decorator** conseguimos verificar uma propriedade de um objeto;
- Vamos criar uma **validação de número máximo de caracteres** com decorators;
- Vamos ver na prática!

Decorators

Conclusão da seção

TS com React

Introdução da seção

React com TS

- Adicionar **TypeScript ao React** nos dá mais possibilidades;
- Seguindo a mesma linha de que em JS, **temos uma forma mais padronizada** para programar;
- Como **tipos para componentes** ou mapeamento de **props por meio de interface**;
- Isso dá mais **confiabilidade** ao projeto e está sendo cada vez mais utilizado hoje em dia;

Instalando React com TS

- Para instalar o **TS junto do React** é simples;
- Vamos começar com **create-react-app** e adicionar a flag **–template** com o valor de typescript;
- **Um novo projeto é criado**, agora com arquivos **.tsx**;
- Podemos inicializá-lo normalmente;
- Vamos ver na prática!

Estrutura de React com TS

- A estrutura de React quando adicionamos TS **não muda muito**;
- Temos as pastas clássicas como: **node_modules**, **src** e **public**;
- Em src que as coisas ficam diferentes, temos a criação de **arquivos .tsx**;
- **Que são arquivos jsx** porém com a possibilidade de aplicação das funcionalidades de TS;
- Podemos executar o projeto com **npm run start**;
- Vamos ver estes arquivos!

Criação de variáveis em componentes

- Podemos **criar variáveis** dentro dos componentes;
- E elas podem receber os tipos que já vimos até este momento do curso;
- Isso nos permite **trabalhar com JSX** com apoio destas variáveis e seus tipos;
- Vamos ver na prática!

Criação de funções em componentes

- Podemos também criar **funções em componentes**;
- Estas funções recebem **parâmetros**, que **podem ser tipados**;
- E o seu retorno também;
- Ou seja, podemos aplicar os mesmos conceitos que já vimos de TS;
- Vamos ver na prática!

Criação de novos componentes

- Geralmente criamos os componentes em uma pasta chamada **components**;
- O arquivo deve ser **.tsx**;
- E um retorno comum utilizado é o **JSX.Element**;
- O resto fica semelhante ao React sem TS;
- Vamos ver na prática!

Extensão para React com TS

- A extensão que vamos utilizar é a **TypeScript React code snippets**;
- Ela nos ajuda com **atalhos** para programar mais rápido;
- Como o **tsrafce**, que cria um componente funcional;
- Isso torna o nosso dia a dia mais simples;
- Vamos ver na prática!

Importando componentes

- A importação de componente **funciona da mesma forma que sem TypeScript**;
- Porém temos que nos atentar aos **valores e tipos das props** de cada componente;
- O TS interage de forma mais sucinta na parte da importação;
- Vamos ver na prática!

Destructuring nas props

- O **destructuring** é um recurso de ES6, que **permite separar um array ou objeto** em várias variáveis;
- Aplicamos esta técnica nas **props**, para não precisa repetir o nome do objeto sempre;
- Podemos fazer desta maneira em TS também;
- Vamos ver na prática!

O hook useState

- o **useState** é um hook muito utilizado em React;
- Serve para **consultar e alterar o estado** de algum dado;
- **Atrelamos uma função set a um evento**, como mudança de dado em input e alteramos o valor da variável base;
- Podemos adaptar este recurso para TS também;
- Vamos ver na prática!

Enum

- O **Enum** é uma forma interessante de formatar um objeto com chaves e valores;
- Onde podemos **utilizar como props**;
- Passando a chave pela prop, imprimimos o valor dela no componente;
- Vamos ver na prática!

Types

- Além das interfaces, podemos criar estruturas de tipos com o **type**;
- Isso nos permite criar dados com **tipos de dados fixos**;
- Ou até tipos customizados, como quando utilizamos o **operador |**
- Vamos ver na prática!

Context API

- A **Context API**, é uma forma de transmitir dados entre componentes no React;
- A ideia principal é que podemos **determinar quais componentes recebem estes dados**;
- Ou seja, fazem parte do **contexto**;
- Podemos aplicar TS a esta funcionalidade também;
- Vamos ver na prática!

Utilizando o dado de contexto

- Para utilizar os dados do contexto vamos precisar de um **hook**;
- Que é o **useContext**;
- A partir dele conseguimos **extrair os dados** e utilizar em um componente;
- Vamos ver na prática!

TS com React

Conclusão da seção

TS com Express

Introdução da seção

Inicialização

- Para iniciar um projeto com Express e TypeScript precisamos criar o projeto com **npm init**;
- E também iniciar o TS com **npx tsc –init**;
- Após estes dois passos **vamos instalar as dependências**, algumas são de de dev (como os tipos) e outras não (como o Express);
- E por fim criamos um **script** e iniciamos a aplicação!

Utilizando o Express

- Para utilizar o express vamos **importar o pacote**;
- E criar ativá-lo em uma nova variável, geralmente chamada de **app**;
- Podemos **criar uma rota** que retorna uma mensagem;
- Definir uma **porta** para a aplicação;
- E verificar o resultado no navegador;

Roteamento

- Podemos utilizar **qualquer verbo HTTP nas rotas** do Express;
- Vamos criar uma que funciona a base de **POST**;
- Para isso precisamos trafegar dados em **JSON**, podemos fazer isso ativando um **middleware**;
- Iremos realizar os testes com o **Postman**;

Rota para qualquer verbo

- Utilizando o método **all**, podemos criar uma rota que aceita qualquer verbo;
- É interessante para quando um endpoint **precisa realizar várias funções**;
- Podemos criar um tratamento para entregar a resposta;
- Vamos ver na prática!

Interfaces do Express

- Para alinhar nossa aplicação ao **TypeScript** vamos adicionar novos tipos;
- As request podem utilizar o tipo **Request**;
- E as respostas o **Response**;
- Vamos ver na prática!

JSON como respostas

- Na maioria das vezes enviamos **JSON para endpoints de API**;
- Para fazer isso com Express é facil!
- Basta enviar o JSON no **método json** de response;
- Vamos ver na prática!

Router parameters

- Podemos pegar parâmetros de rotas com Express;
- Vamos utilizar **req.params**;
- A rota a ser criada precisa ser **dinâmica**;
- Ou seja, os parâmetros que estamos querendo receber precisam estar no padrão: **:id**;

Rotas mais complexas

- Podemos ter rotas com **mais de um parâmetro**;
- Todos os dados continuam em **req.params**;
- O padrão é: /algo/**:param1**/outracoisa/**:param2**
- Teremos então: param1 e param2 em req;
- Vamos ver na prática!

Router handler

- **Router handler** é um recurso muito interessante para o Express;
- Basicamente retiramos a função anônima de uma rota e **externalizamos em uma função**;
- Podemos reaproveitar essa função, ou estrutura nossa aplicação desta maneira;
- Vamos ver na prática!

Middleware

- **Middleware** é um recurso para executar uma função entre a execução de uma rota, por exemplo;
- O nosso **app.use de json** é um middleware;
- Podemos utilizar middleware para **validações**, por exemplo;
- Vamos ver na prática!

Middleware para todas as rotas

- Para criar um middleware que é executado em todas as rotas vamos utilizar o método **use**;
- **Criamos uma função** e atrelamos ao método;
- Desta maneira **todas as rotas** terão ação do nosso middleware;
- Vamos ver na prática!

Request e Response generics

- Podemos estabelecer os **argumentos que vem pelo request e vão pela response**;
- Para isso vamos utilizar os **Generics** de Response e Request;
- Que são as **Interfaces** disponibilizadas pelo Express;
- Vamos ver na prática!

Tratando erros

- Para tratar possíveis erros utilizamos **blocos try catch**;
- Desta maneira **podemos detectar algum problema** e retornar uma resposta para o usuário;
- Ou até mesmo **criar um log** no sistema;
- Vamos ver na prática!

TS com Express

Conclusão da seção