PET Full Stack - Notas de Aula

Aula 1

Objetivos

Esta aula tem como objetivo familiarizar o aluno com o básico do desenvolvimento web:

- como funciona e conceitos básicos;
- como escrever um *website* bem simples;
- como pesquisar informação na *internet*.

Ambiente de desenvolvimento

Um *website* é construído em cima de três linguagens: HTML, CSS e JavaScript. Essas três serão melhor detalhadas no tópico seguinte, mas antes de mais nada é importante entender que **o navegador entende essas linguagens nativamente**. Ou seja, para desenvolver com essas linguagens basta possuir um navegador de *internet*.

Editores de código

Apesar de não ser necessário, é recomendado possuir instalado um editor de código, como:

- Visual Studio Code (não confundir com Visual Studio)
- Atom
- Sublime Text
- Notepad++
- Vim

Para desenvolvimento *web*, é muito comum a utilização do editor Visual Studio Code por conta das ferramentas e extensões que ele provê, mas nesse começo de aprendizado muitos desses utensílios podem não fazer sentido e desviar o foco, então escolha o editor que estiver acostumado.

Abrindo arquivos HTML com o navegador

Crie, através do editor de código, um arquivo com extensão `.html` e insira o seguinte código:

```
<html>
    <body><h1>Hello, world!</h1></body>
</html>
```

Abra o arquivo com seu navegador e preste atenção na URL. Perceba que ela começa com `file://` ao invés de `http://`, como é comum em páginas *web*. Esse é o *URL scheme*, e serve para identificar a "maneira" como o navegador está recebendo um recurso. Isso ficará mais claro no futuro, apenas entenda que há diferenças entre abrir um arquivo `.html` diretamente no navegador, e utilizar o protocolo HTTP.

O que aconteceu aqui foi que o navegador acessou o arquivo através da URL, recebeu os dados do arquivo, interpretou e exibiu `Hello, world!`, o que significa que de fato, o navegador é capaz de interpretar HTML nativamente.

No entanto, para interpretar código CSS ou JavaScript no navegador, é preciso colocá-lo "dentro" de um documento HTML. O arquivo HTML é o ponto de entrada de um *website*.

Bônus: Depois desta aula, tente abrir um arquivo `.js` diretamente pelo navegador e perceba que o código é apenas mostrado, mas não executado. Como foi dito, para que o navegador interprete código CSS ou JavaScript, é necessário que o código esteja "dentro" de um código HTML. No entanto, a maioria dos navegadores atuais são capazes de abrir diretamente outros tipos de arquivo, como imagens e PDFs. Com isso, é possível utilizar um navegador no lugar do visualizador de imagens padrão do sistema operacional, por exemplo.

DevTools

Praticamente todo navegador *web* provê "ferramentas de desenvolvedor" (também chamadas de DevTools). Essas ferramentas são extremamente úteis para *debug*, podendo ser utilizadas não somente para o código HTML, CSS e JavaScript, mas também para rede e segurança. Aos poucos, nesta aula, entenderemos como usar algumas dessas ferramentas.

Overview das linguagens HTML, CSS e JavaScript

As três linguagens se completam no desenvolvimento de uma página *web*.

- HTML serve para definir o esqueleto da página. Nele definimos os elementos presentes e a hierarquia entre eles.
- CSS serve para definir a estilização dos elementos, "deixar o *site* bonito".
- JavaScript serve para programar, escrevendo algoritmos que alteram o comportamento padrão da página.

Vamos alterar o código anterior e colocar um pouco de código CSS.

```
<html>
    <head>
        <style>
            h1 {
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>Hello, world!</h1>
    </body>
</html>
```

Observação: O código dentro da *tag* `style` não é HTML, e sim CSS.

Observação 2: Indentações e espaços entre as *tags* (o que está entre os símbolos `<` e `>`) não influenciam no resultado da página.

Salve e atualize a página no navegador. Perceba que agora o texto `Hello, world!` está em vermelho. Apenas reforçando: código CSS serve para estilizar a página.

Vamos adicionar agora um botão e um código JavaScript para acrescentar comportamento ao botão.

```
<html>
    <head>
        <style>
            h1 {
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>Hello, world!</h1>
        <button id="botao">Clique em mim</button>
        <script>
            var botao = document.getElementById("botao");
            botao.onclick = function() {
                alert("Clicado!");
            }
        </script>
    </body>
</html>
```

Observação: O código dentro da *tag* `script` não é HTML, e sim JavaScript.

Observação 2: Perceba que o código JavaScript precisou ser adicionado após a definição do botão. Caso contrário, não seria possível acessar esse elemento com `document.getElementById`. Perceba também que, com a *tag* `style`, não houve essa preocupação.

A ideia desse tópico era apresentar as três linguagens e assimilar a função de cada uma delas. Não se preocupe, cada linguagem será abordada com mais detalhes a partir de agora.

HTML

Vamos falar brevemente sobre a linguagem HTML.

Antes de mais nada, vamos utilizar a versão com mais *features*, o HTML5. Para utilizar HTML5 no seu código, basta adicionar ao topo do arquivo, na primeira linha: `<!DOCTYPE html>`.

Função da linguagem

Como dito anteriormente, HTML é uma linguagem que serve para definir o esqueleto de uma página *web*. Mais detalhadamente, utilizamos a linguagem HTML para definir como será nosso DOM (Document Object Model).

O DOM é um modelo, em forma de árvore, que armazena todas as informações para renderizar uma página *web*. Cada nó dessa árvore é chamado de "DOM Element", e possui uma relação direta com as *tags* do HTML. O navegador interpreta o código HTML e converte cada *tag* em um elemento do DOM, por isso dizemos que a linguagem HTML é uma linguagem de **marcação**.

Cada elemento DOM possui vários atributos que podem ser alterados via JavaScript ou HTML. Já a linguagem CSS foi desenhada para manipular principalmente o atributo `style`.

Estrutura básica

Mesmo com um código HTML em branco, o navegador cria no DOM ao menos os elementos `html`, `head` e `body`. Isso pode ser verificado abrindo as DevTools e verificando os elementos da página. Esses elementos são a base de toda página *web*.

Segue a função dos elementos:

- `html`: É a raiz do DOM. Todos os outros elementos devem ser filhos deste.
- `head`: Deve possuir como filhos elementos que **não são renderizados**.
- `body`: Deve possuir como filhos elementos que **são renderizados**.

Observação: Nas DevTools é comum utilizar código HTML para descrever o DOM, afinal, facilita a leitura do desenvolvedor.

Bônus: No tópico anterior utilizamos a *tag* `script` dentro do `body`. Com as novas *features* do HTML5, é possível colocar a *tag* `script` no `head` e acrescentar o atributo `defer`. Esse atributo fará com que o código JavaScript seja executado após a construção do DOM, mas para funcionar corretamente, deve ser utilizado um arquivo externo com extensão `.js`.

Tags HTML

Como foi dito anteriormente, em geral cada *tag* HTML corresponde a um elemento DOM. O *website* https://allthetags.com/ possui informações sobre várias delas.

É possível que algumas *tags* não sejam suportadas por versões mais antigas do HTML e dos navegadores. Nesse caso, o *website* https://caniuse.com/ se mostra muito útil. Nele contém informação sobre quais navegadores suportam determinada *feature*, não só do HTML mas do CSS e do JavaScript também. Além disso, o *site* informa sobre quantas pessoas no mundo utilizam determinada versão de um navegador.

Em geral, a sintaxe de uma *tag* pode ser encontrada em uma das seguintes formas:

```
<!-- Padrão -->
<nomeDoElemento atributo1="valor1" atributo2="valor2" atributoBooleano1
atributoBooleano2>
    Conteúdo interno (inner HTML)
    <tagInterna>...</tagInterna>
</nomeDoElemento>

<!-- Self closing -->
<nomeDoElemento atributo1="valor1" atributo2="valor2" atributoBooleano1
atributoBooleano2 />
```

Entre os símbolos `<>` deve ser informado primeiramente qual elemento será criado no DOM, como por exemplo, `p` para criar um parágrafo. Em seguida, informamos como os atributos padrão do elemento serão sobrescritos.

Sobre os atributos, é importante notar duas coisas:

- Elementos diferentes, em geral, possuem conjuntos de atributos diferentes. Na *tag* `img` (*image*), por exemplo, é comum utilizar o atributo `src` para definir o local em que aquela imagem pode ser encontrada. Esse atributo não faz sentido para a *tag* `p` (*paragraph*), que serve para lidar com texto.
- A maioria dos atributos podem receber uma "*string*" como valor, com exceção dos atributos booleanos. A presença deles indica *true*, e a ausência *false*, não se engane fazendo algo como `atributoBooleano="false"`. O navegador entenderá isso como *true*.

Importante: Grande parte das *tags* podem ser "abertas" (`<>`) e "fechadas" (`</>`). Esse tipo de *tag* pode possuir filhos na árvore de elementos DOM, e esses filhos devem ser colocados entre a abertura e o fechamento da *tag*. Nesse exemplo, o elemento referente à primeira *tag* possui como filhos um *text node* e um elemento do tipo `tagInterna`. Enquanto isso, as *tags* do tipo *self closing* não possuem filhos.

Observação: O conteúdo dentro de `<!--` e `-->` é um comentário, ou seja, não será interpretado pelo navegador, mas ajuda a organizarmos o código.

Um pouco de prática

Dominar esses conceitos iniciais é importantíssimo para saber o que acontece por "baixo dos panos", evitando *bugs*, e para saber como pesquisar e ler informação na internet.

No entanto, a prática é essencial, não apenas para fixar os conceitos aprendidos, mas também para se familiarizar com as *tags* e seus atributos.

Vamos criar um *tweet* simples utilizando HTML.

```
<div>

    <header>
        <img src="img/neymar.jpg" />
        <h2>Neymar Jr</h2>
        <h3>@neymarjr</h3>
    </header>

    <p>
        Bora falar de bbb porque de futebol esse fds não foi legal 😢
    </p>

    <footer>
        <p>
            6:14 p. m. - 31 jan. 2021 - Twitter for iPhone
        </p>
    </footer>

</div>
```

Note que, como ainda não aprendemos CSS, não será possível estilizar o *tweet*. Foque, por enquanto, na estrutura básica.

Segue uma breve explicação de cada um dos elementos:

- `div`: Uma divisão no *layout*, serve para agrupar elementos relacionados (terá uma utilidade maior no CSS).
- `header` e `footer`: Mesma coisa que `div`, mas com um significado semântico. Facilita um pouco a leitura do código e o entendimento dos mecanismos de busca (Google, Bing,

DuckDuckGo, etc) sobre a sua página.
- `img`: Imagem cujo atributo `src` contém o local em que ela pode ser encontrada.
- `h2` e `h3`: Hierarquia de títulos e subtítulos (juntamente com `h1`, `h4`, `h5` e `h6`)
- `p`: Trecho de texto, não necessariamente utilizado literalmente para parágrafos textuais.

Observação: Neste código estão sendo omitidas as *tags* `html` e `body`, mas elas devem estar presentes no seu código, bem como a indicação de uso do HTML5: `<!DOCTYPE html>`.

Observação 2: É quase certo que este código HTML precisará ser alterado quando introduzirmos o CSS.

Referência para recursos

Como foi visto no último código, podemos utilizar o atributo `src` da *tag* `img` para referenciar um arquivo do tipo imagem. Perceba que para que o código funcione adequadamente, a pasta `img` deve estar na mesma pasta que o nosso arquivo HTML, e deve existir dentro de `img` um arquivo de imagem chamado `neymar.jpg`.

Isso tudo porque **estamos utilizando o sistema de arquivos** para navegar entre os recursos. Nesse caso, `img/neymar.jpg` é equivalente a `./img/neymar.jpg`. Já se utilizarmos `/img/neymar.jpg`, teremos uma referência para a raiz do sistema de arquivos, seguida de uma pasta `img` e um arquivo de imagem.

Guarde este exemplo na memória. Ainda nesta aula, utilizaremos o protocolo HTTP e haverá grandes diferenças quanto a essas referências.

Apenas para fixar, vamos criar um novo arquivo HTML dentro da mesma pasta que o arquivo inicial, em `views/about.html`. Era comum em *websites* mais antigos, como *blogs*, a presença de uma página "Sobre", que continha informações sobre o criador.

Vamos utilizar também a *tag* `a` e o atributo `href` para criar um *hyperlink* entre a página principal e a página *about*.

```
<!-- arquivo principal -->
<a href="views/about.html">Sobre</a>
```

Vamos fazer o mesmo para a página *about*, supondo que o nome do arquivo da página principal seja `index.html`.

```
<!-- about.html -->
<a href="../index.html">Início</a>
```

Novamente, isso funciona pois estamos utilizando o sistema de arquivos.

Escrevendo código HTML

Resumidamente, ao escrever código HTML você deve se preocupar principalmente com três coisas:

- Quais elementos devem estar presentes no *site*.
- Qual a hierarquia entre eles.
- Estruturar o layout de modo a facilitar o código CSS.

Há uma quarta preocupação quanto à semântica dos elementos, como o uso de `header` e `footer` no lugar de `div`, mas não é algo tão importante, especialmente nesse começo de aprendizado.

É importante reforçar que **não se estiliza páginas com HTML**. Um erro muito comum de *devs* iniciantes é utilizar as *tags* `h1 .. h6` baseadas em seu tamanho, sendo que este pode ser alterado facilmente via CSS. Essas *tags* devem ser utilizada apenas para estabelecer uma hierarquia entre os textos de sua página, o que significa que o uso delas tem mais a ver com a semântica do *site*.

Observação: Note que no exemplo do *tweet* foi utilizada a *tag* `h2`. Isso porque é boa prática cada página possuir um único `h1`, e como espera-se que hajam vários *tweets* dentro de um *site*, optei pela utilização do `h2`.

CSS

O código CSS, como foi dito anteriormente, serve para estilizar a página. Com ele, vamos fazer o nosso *tweet* criado ficar "com cara" de *tweet*.

Onde escrever CSS

Há 3 opções:

- direto no atributo `style` do elemento, o que é extremamente não recomendado.

```
<p style="color: red">Texto vermelho</p>
```

- dentro de uma *tag* `style`.

```
<head>
    <style>
        p {
            color: red;
        }
    </style>
</head>
<body>
    <p>Todos os parágrafos</p>
    <p>Ficaram vermelhos</p>
</body>
```

- dentro de um arquivo de extensão `.css`, em conjunto com a *tag* `link`.

```
<!-- index.html -->
<head>
    <link rel="stylesheet" href="styles.css" />
</head>
<body>
    <p>Todos os parágrafos</p>
    <p>Ficaram vermelhos</p>
</body>
```

```css
/* styles.css */
p {
    color: red;
}
```

A utilização ou não de um arquivo externo para o CSS depende de bom senso e padrão de projeto. Para códigos pequenos, facilita fazer no próprio arquivo `.html`.

Observação: Projetos maiores costumam ser complicados de desenvolver com HTML e CSS puros. Existem soluções, como *React.js*, que facilitam o desenvolvimento de *websites* complexos.

Observação 2: Note que o comentário em CSS possui uma sintaxe diferente do HTML.

Rules CSS

Em média, mais que 95% do código CSS puro são regras (*rules*). Segue a sintaxe de uma regra:

```
seletor {
    propriedade1: valor1;
    propriedade2: valor2;
    ...
}
```

O "seletor" serve para identificar a quais elementos do DOM essa regra se aplica. As chaves representam o "bloco de declaração". Cada linha terminada em `;` é uma "declaração", composta por "propriedade" e "valor".

Cada conjunto propriedade-valor altera um aspecto da estilização dos elementos selecionados pelo seletor.

Id e Class

Suponhamos em nosso exemplo anterior dos parágrafos que queremos que apenas um parágrafo particular fique da cor vermelho. Podemos definir um atributo `id` para ele e selecioná-lo via CSS.

```html
<!-- index.html -->
<p>Nem todos os parágrafos</p>
<p id="vermelho">Ficaram vermelhos</p>
```

```css
/* styles.css */
#vermelho {
    color: red;
}
```

Vamos supor agora que temos 5 parágrafos, e queremos que apenas o primeiro e o terceiro fiquem vermelhos. Podemos utilizar para isso o atributo `class`. A principal diferença entre os dois é que pode haver múltiplos elementos da mesma classe, enquanto o id deve ser associado a um único elemento dentro da página. Além disso, um elemento pode possuir múltiplas classes, mas somente um id.

```html
<!-- index.html -->
<p class="vermelho">Parágrafo 1</p>
<p>Parágrafo 2</p>
<p class="vermelho">Parágrafo 3</p>
<p>Parágrafo 4</p>
<p>Parágrafo 5</p>
```

```css
/* styles.css */
.vermelho {
    color: red;
}
```

Então utilizamos no seletor o caractere

- `#` para `id`
- `.` para `class`
- nada para tipo de elemento

Combinando seletores

Podemos combinar seletores de várias formas. Serão apresentadas três delas:

- Concatenando seletores

  ```html
  <!-- index.html -->
  <p class="vermelho">Parágrafo com classe "vermelho"</p>
  <h1 class="vermelho">Título com classe "vermelho"</p>
  ```

  ```css
  /* styles.css */
  h1.vermelho {
      color: red;
  }
  ```

  Nesse caso, as regras se aplicarão apenas àqueles elementos que se encaixarem em todos os seletores concatenados.

- Colocando vírgulas entre os seletores

  ```html
  <!-- index.html -->
  <p>Parágrafo</p>
  <h1>Título</p>
  ```

  ```css
  /* styles.css */
  p, h1 {
      color: red;
  }
  ```

  Nesse caso, as regras se aplicarão aos elementos que se encaixarem em pelo menos um dos seletores separados por vírgula.

- Colocando um espaço entre os seletores

```html
<!-- index.html -->
<div id="paragrafos">
    <h1>Título filho</h1>
    <p>Parágrafo filho</p>
    <div>
        <p>Parágrafo neto</p>
    </div>
</div>
<p>Parágrafo irmão</p>
```

```css
/* styles.css */
#paragrafos p{
    color: red;
}
```

Nesse caso, selecionamos todos os elementos dentro do primeiro seletor que satisfazem o segundo seletor.

Podemos ainda combinar essas "operações" e criar uma lógica complexa de seleção de elementos.

```css
/* styles.css */
div.vermelho p, h1 {
    color: red;
}
```

Observação: O assunto "seletor" é muito mais profundo do que aquilo que foi apresentado aqui, e há muitas informações importantes que foram deixadas de lado por enquanto. O tempo é curto e infelizmente não dá para abordar tudo agora, no entanto, as ferramentas apresentadas já são bem poderosas.

DevTools - Elements

Com as ferramentas de desenvolvedor, na aba "*elements*", pode ser verificado o que está acontecendo no DOM, verificando quanto espaço na página um determinado elemento está ocupando, e quais conjuntos propriedade-valor foram aplicados a ele. Além disso, é possível manipular os valores do HTML e CSS e fazer testes. Note que as alterações feitas nas ferramentas de desenvolvedor não serão salvas ao recarregar a página.

Containers div e span

Containers, em HTML, são elementos de *layout* que, essencialmente, armazenam outros elementos e os organiza.

Os principais *containers* são `div` e `span`, e a principal diferença está no modo de `display` deles.

- `div` possui por padrão o valor `block` na propriedade `display`.
- `span` possui por padrão o valor `inline` na propriedade `display`.

Não vamos entrar em detalhes da diferença, vamos nos contentar com como esses elementos são usados na prática:

- `div` você usa para elementos de *layout* quaisquer, dividindo a página em vários componentes, um dentro do outro.

- `span` você usa para estilizar uma parte de um texto, por exemplo para deixar em negrito e na com vermelha.

O uso de `div` é muito mais frequente que de `span`. Pensando em um *blog*, podemos usar `divs` para, por exemplo:

- Dividir a página em: barra de navegação, conteúdo principal e conteúdo lateral.
- Dividir o conteúdo principal em: postagem 1, postagem 2, postagem 3, ...
- Dividir cada postagem em: cabeçalho, conteúdo, rodapé.

A estrutura que acabamos de descrever, usando apenas `divs`, deve se parecer com isso:

```
<div id="navegacao">...</div>
<div id="conteudo-principal">
    <div class="postagem">
        <div class="cabecalho">...</div>
        <div class="conteudo">...</div>
        <div class="rodape">...</div>
    </div>
    <div class="postagem">...</div>
    <div class="postagem">...</div>
    <div class="postagem">...</div>
    <div class="postagem">...</div>
</div>
<div id="conteudo-lateral">...</div>
```

Observação: Apesar deste conteúdo estar fortemente ligado ao HTML, é comum utilizarmos *containers* quando desejamos fazer algo com eles no CSS. Por isso acredito que faz sentido trazer esse conteúdo para cá.

Tamanho, margem, borda e padding

Nesta seção, é muito importante o uso das ferramentas de desenvolvedor.

Vamos brincar um pouco com as `divs`.

```
<!-- index.html -->
<div id="bloco-principal"></div>
```

```
/* styles.css */
#bloco-principal {
    width: 100px;
    height: 100px;
    background: red;
}
```

Perceba que foi criado um quadrado vermelho com 100px de largura. Usando as DevTools, verifique que a margem presente na página vem do elemento `body`. Vamos arrumar isso.

```
/* styles.css */
body {
    margin: 0;
}
```

Agora a margem sumiu. Vamos adicioná-la de volta, mas dessa vez na `div`

```css
/* styles.css */
#bloco-principal {
    ...
    margin: 10px;
}
```

Perceba que apesar de definirmos a margem como 10px, ela se estende à direita até o fim da página. Esse comportamento é consequência do valor `block` na propriedade `display`. Vamos alterar para `inline-block`, dessa forma temos o comportamento esperado.

```css
/* styles.css */
#bloco-principal {
    ...
    display: inline-block;
}
```

Com esse ambiente construído, é hora de entender a diferença entre "margem", "borda" e "padding".

```css
/* styles.css */
#bloco-principal {
    ...
    border: 5px solid black; /* Adicione primeiro esta linha e use as DevTools
*/
    padding: 10px; /* Depois, adicione esta linha e use novamente as DevTools */
}
```

Podemos verificar com esse simples experimento que:

- A margem define um espaçamento entre o elemento e outros elementos
- O padding define um espaçamento entre o elemento e seu conteúdo
- A borda representa a fronteira do elemento.
- Cada um dos três é adicionado individualmente (a largura final do quadrado é 10 + 5 + 10 + 100 + 10 + 5 + 10 = 150px)

Agora vamos verificar tamanho

```html
<!-- index.html -->
<div id="bloco-principal">
    <div id="bloco-interno"></div>
</div>
```

```css
/* styles.css */
#bloco-interno {
    width: 30px;
    height: 30px;
    background: blue;
    display: inline-block;
}
```

O que acha que acontece quando definimos a largura do bloco interno para `100%`? Vamos testar

```css
/* styles.css */
#bloco-interno {
    width: 100%;
    ...
}
```

Preencheu toda a largura do bloco principal, o que significa que % nesse contexto é em relação ao tamanho do conteúdo do pai. Experimente aumentar para `200%` e veja que ocorre o que chamamos de `overflow`, que é quando um elemento filho fica maior do que o pai.

Para utilizarmos uma porcentagem com relação ao tamanho da página, devemos usar `vw` para largura e `vh` para altura.

```css
/* styles.css */
#bloco-interno {
    width: 50vw;
    ...
}
```

Perceba que, redimensionando a página, a proporção é mantida.

Flexbox

Flexbox é uma das tecnologias mais simples e mais poderosas no desenvolvimento de *layouts*. Para utilizá-la em uma `div`, basta definir a propriedade `display` como `flex`, sobrescrevendo o valor padrão `block`.

```html
<!-- index.html -->
<div>
    <p>Me centralize!</p>
    <p>Me centralize também!</p>
</div>
```

```css
/* styles.css */
div {
    width: 100%;
    height: 100%;
    background: lightcoral;
    display: flex;
}
```

Veja como é fácil centralizar conteúdos com *flexbox*.

```css
/* styles.css */
div {
    ...
    justify-content: center;
}
```

Mas queremos que os parágrafos apareçam na forma de coluna.

```css
/* styles.css */
div {
    ...
    flex-direction: column;
}
```

Agora não está mais alinhado ao centro! Isso porque a propriedade `justify-content` centraliza com relação ao eixo principal. Para centralizar com relação ao eixo secundário, é preciso utilizar a propriedade `align-items`.

```css
/* styles.css */
div {
    ...
    align-items: center;
}
```

Caso tenha ficado com dúvida em alguma etapa, utilize as ferramentas de desenvolvedor.

É de extrema importância dominar *flexbox*, pois é aplicável em diversas partes de um *layout*, e muito simples de utilizar. As DevTools do Google Chrome fornecem um jeito muito simples de testar o comportamento de *flexbox*.

Segue um conteúdo complementar sobre *flexbox*, do canal Cod3r Cursos: https://www.youtube.com/watch?v=s-CARPA01NU. Não se preocupe com as tecnologias citadas durante o vídeo, foque no essencial.

Conhecendo outras propriedades e valores

Não daremos muito enfoque nos conjuntos propriedade-valor, pois não vale a pena. Nessa situação, o que mais conta é a experiência, e para conseguir isso, pense sempre no "o que fazer" antes do "como fazer".

Ou seja, pense em um elemento, como por exemplo um parágrafo, e pense como deseja estilizá-lo. Depois, procure as propriedades que podem fazê-lo atingir esse objetivo. Se necessário, consulte a *internet* no processo.

Pode parecer óbvio esse processo de primeiro pensar "o que fazer" para depois "como fazer", mas não é. O que acontece é que, conhecendo algumas poucas propriedades, você pode ficar estagnado e limitando-se ao que você sabe, seria "fazer aquilo que você sabe que consegue". Isso é péssimo pro aprendizado, principalmente porque grande parte do conteúdo de CSS puro na internet é obsoleto, e isso pode te fazer pensar que aquilo é o máximo que você pode fazer com CSS.

Porém, CSS é uma linguagem muito poderosa, e dá pra fazer muita coisa legal, bonita e moderna com ela, basta procurar e aprender. Caso tenha dúvida, navegue no *site* https://codepen.io/.

Observação: Tenha o bom senso de não escolher algo muito complicado para fazer logo de início. No processo de aprendizagem, é importante fixar conceitos utilizando códigos pequenos.

Estilizando o Tweet

Hora de estilizar o *tweet* cujo HTML foi feito anteriormente.

```html
<!-- index.html -->
<div class="tweet">
    <header>
```

```
        <img src="img/neymar.jpg" />
        <div class="names">
            <h2>Neymar Jr</h2>
            <h3>@neymarjr</h3>
        </div>
    </header>
    <p class="main-text">
        Bora falar de bbb porque de futebol esse fds não foi legal 😢
    </p>
    <footer>
        <p>6:14 p. m. - 31 jan. 2021 - Twitter for iPhone</p>
    </footer>
</div>
```

```
/* styles.css */
body {
  margin: 0;
  display: flex;
  flex-direction: column;
  align-items: center;
}

p,
h2,
h3 {
  margin: 0;
  font-family: Arial, Helvetica, sans-serif;
}

.tweet {
  margin-top: 30px;
  display: flex;
  flex-direction: column;
  max-width: 500px;
}

.tweet header {
  display: flex;
}

.tweet header .names {
  display: flex;
  flex-direction: column;
  margin-left: 10px;
  justify-content: center;
}

.tweet header .names h2 {
  margin-bottom: 3px;
  font-size: 15px;
  font-weight: 800;
}

.tweet header .names h3 {
  font-size: 14px;
  font-weight: 500;
  color: #666;
```

```
}

.tweet header img {
  width: 60px;
  height: 60px;
  object-fit: cover;
  border-radius: 100%;
}

.tweet .main-text {
  margin: 10px 0;
  font-size: 20px;
}

.tweet footer p {
  color: #666;
}
```

Perceba como o código HTML teve de ser adaptado.

JavaScript

Vamos fazer uma breve introdução à linguagem de programação JavaScript.

Onde escrever JavaScript

Coloque dentro do `head` a *tag* `script`.

```
<head>
    <script>
        alert("Olá, mundo!");
    </script>
</head>
```

Um pouco sobre a linguagem

A linguagem JavaScript possui vários dos recursos mais simples de outras linguagens, como variáveis, `if`, `for`, `while`, mudando apenas a sintaxe em algumas situações.

É uma linguagem cheia de problemas e particularidades, mas muito poderosa caso sejam seguidas as boas práticas. Isso não será abordado nessa aula por ser um conteúdo bem extenso.

O código é executado linha por linha de forma síncrona. No entanto, é possível adicionar elementos de assincronismo, com eventos.

```
setTimeout(function() {
    alert("Olá, mundo!");
}, 5000);
```

A função anônima (sem nome) será executada após, pelo menos, 5000 milissegundos (5 segundos). Quando essa função anônima é executada, é exibida na tela a mensagem "Olá, mundo!".

Acessando o DOM

O navegador fornece APIs para manipular uma página via JavaScript. Uma das mais importantes APIs é o objeto `document`. Com ele podemos acessar elementos do DOM via id, por exemplo.

```
<head>
    <script>
        // A variável "paragrafo" armazena um elemento DOM
        var paragrafo = document.getElementById("paragrafo");

        paragrafo.innerHTML += " <3";
        paragrafo.style.color = "red";
        paragrafo.onclick = function() {
            alert("Elemento clicado");
        }
    </script>
</head>
<body>
    <p id="paragrafo">Isto é um parágrafo manipulado via JavaScript!</p>
</body>
```

Há um problema com esse código: o `script` está sendo executado antes da construção completa do DOM. Para que o código funcione devidamente há duas opções: colocar o `script` ao fim do `body` ou adicionar o atributo booleano `defer` e utilizar um arquivo externo. Vamos seguir com a última opção.

```
<!-- index.html -->
<head>
    <script src="main.js" defer></script>
</head>
<body>
    <p id="paragrafo">Isto é um parágrafo manipulado via JavaScript!</p>
</body>
```

```
// main.js
var paragrafo = document.getElementById("paragrafo");

paragrafo.innerHTML += " <3";
paragrafo.style.color = "red";
paragrafo.onclick = function() {
    alert("Elemento clicado");
}
```

Agora, caso esteja utilizando HTML5, a página se comportará como esperado.

Observação: Perceba que o atributo `onclick` executa uma função anônima cada vez que o elemento em questão é clicado. Os atributos que começam com `on` são chamados de "eventos" e são capazes de executar uma função após algo ter ocorrido com relação ao elemento.

DevTools - Console

A aba *console* das DevTools nos fornece informações de erros no JavaScript e na obtenção de recursos, além de ser possível visualizar o conteúdo dentro de uma variável e alterar o comportamento de um elemento.

Para exibir variáveis ou objetos no *console*, use a função `console.log`. Note que `console`, assim como `document`, se trata de outra API fornecida pelo navegador.

Sistemas Web

Vamos estudar o funcionamento de um sistema *web* simples, com um servidor responsável por entregar páginas estáticas ao cliente.

Servidor

Um servidor nada mais é um processo que está aberto para comunicação com outros processos. Vamos estudar os servidores que seguem o modelo de *request* e *response*.

Segue uma **ANALOGIA**.

Pense em um programa cuja função é receber um inteiro como entrada e produzir como saída o dobro deste número, em *loop*. Seria algo como:

```
while (1) {
    int num;
    scanf("%d", &num);
    printf("%d\n", 2 * num);
}
```

Este é o nosso "servidor". Vamos executá-lo

```
> 2    (request)
< 4    (response)
> 10
< 20
> 313
< 626
```

Você, como usuário, fez papel de **CLIENTE**. Você manda *requests* contendo um número ao servidor, e o servidor te manda uma *response* contendo o dobro desse número.

Pense agora em outro programa que é "cliente" deste primeiro. Esse programa envia um número ao servidor e recebe o dobro deste número como resposta, depois salva em um arquivo.

```
FILE *fp;
...
int num = 3;
int dobro;
printf("%d\n", num);
scanf("%d", &dobro);
fprintf(fp, "%d\n", dobro);
```

Vamos executá-lo.

```
< 3    (request)
> 6    (response)
~ salva valor 6 em um arquivo ~
```

Perceba que agora, você como usuário fez papel de **SERVIDOR**. Quem recebeu a *request* foi você, e devolveu como *response* o dobro do número informado.

Pense agora que existe uma maneira de fazer com que os dois programas se comuniquem. Não vamos escrever a implementação em C, mas assuma que isso é possível.

```
+--------------+                +--------------+
|              |     Como       |              |
|  server.exe  |  estabelecer   |  client.exe  |
|              |  comunicação?  |              |
+--------------+                +--------------+
```

Podemos usar as portas de rede para isso. Vincularemos o processo `server.exe` à porta 3000.

```
+--------------+                +--------------+
|              +------+         |              |
|  server.exe  | 3000 |         |  client.exe  |
|              +------+         |              |
+--------------+                +--------------+
```

Podemos agora, fazer com que o `client.exe`, mande requisições ao `server.exe`.

```
+--------------+                +--------------+
|              +------+         |              |
|  server.exe  | 3000 |<------->|  client.exe  |
|              +------+         |              |
+--------------+                +--------------+
```

Nessa situação, o nosso servidor:

- Fica aguardando requisições na porta 3000.
- Quando recebe um número como *request*, retorna como *response* o dobro do número informado ao `client.exe`.

Note que neste modelo uma ***response* necessariamente ocorre após uma *request***.

Do ponto de vista do cliente:

- Mandamos um número para a porta 3000, como *request*.
- Recebemos como *response* um número, que seria o dobro daquele informado na *request*.
- Armazenamos esse número em um arquivo.

E o que acontece quando enviamos como *request* algo que não é número, como uma *string*? Para que tudo ocorra bem, deve ser estabelecido um padrão de comunicação entre cliente e servidor. É aí que entra o protocolo HTTP.

Protocolo HTTP

O protocolo HTTP nada mais é que uma linguagem de comunicação entre um cliente e um servidor. Possui como principal objetivo a entrega de recursos (páginas *web*, imagens, arquivos, etc.) na rede, e utiliza o modelo *request/response*.

Sempre que você escreve um URL no navegador e aperta *enter*, o navegador envia uma *request* seguindo este formato:

```
GET /index.html HTTP/1.1
Host: www.meusite.com
Accept-Language: pt-br
```

E recebe do servidor uma *response*:

```
HTTP/1.1 200 OK
Date: 1 Jan 2021 09:00
Content-Type: text/html

<!DOCTYPE html>
<html>
...
</html>
```

Cada *request/response* HTTP é dividida em *header* e *body*. No caso de requisições GET, não há *body*.

O protocolo HTTP segue o mesmo diagrama feito anteriormente, a diferença é que, no caso de um *website* já devidamente hospedado, a internet faz o intermédio na comunicação entre o processo cliente e o servidor. No lugar de `client.exe` seria `chrome.exe` por exemplo, caso o navegador em questão seja o Google Chrome.

Criando um servidor HTTP

Node.js é um programa que roda JavaScript fora do navegador. Com ele e o *framework* Express, podemos criar um servidor HTTP de forma bem simples.

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

app.get('/', function (request, response) {
  response.send('Hello, world')
})

app.listen(3000)
```

Neste código, criamos um servidor HTTP que, sempre que recebe uma requisição na rota raiz, retorna "Hello World" no *body* da *response*. Os *response headers* nesse caso são gerados automaticamente.

Para enviar essa requisição basta colocar no URL do navegador `http://localhost:3000` e apertar *enter*. `localhost` serve para referenciar a própria máquina.

Rotas

O conceito de "rota" é muito importante em sistemas *web*. Quando definimos `app.get('/', ...)` estamos utilizando a rota raiz, como se fosse um sistema de arquivos. Se alterarmos para `app.get('/indice', ...)`, por exemplo, e acessarmos o mesmo URL de antes, obteremos o erro *not found* (código 404).

```
HTTP/1.1 404 Not Found
X-Powered-By: Express
...
```

Então devemos acessar o URL `http://localhost:3000/indice` para tudo funcionar corretamente.

Vamos agora enviar uma página HTML através do servidor.

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

app.get('/', function (request, response) {
  response.sendFile(path.join(__dirname, 'index.html'))
})

app.listen(3000)
```

Vamos criar, dentro da mesma pasta do servidor, o arquivo `index.html`.

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <body>
    <p>Documento HTML genérico</p>
    <a href="views/other.html">Link para outro documento HTML.</a>
  </body>
</html>
```

E, dentro de `views`, vamos criar o arquivo `other.html` e preencher com o conteúdo:

```
<!-- other.html -->
<!DOCTYPE html>
<html>
  <body>
    <p>Outro documento HTML genérico</p>
    <a href="../index.html">Voltar</a>
  </body>
</html>
```

Como fizemos anteriormente. Deve funcionar, certo?

Ao clicar no hyperlink `Link para outro documento HTML`, somos redirecionados para a rota `http://localhost:3000/views/other.html`.

Essa rota não está definida, e por isso o servidor envia *not found* com a mensagem `Cannot GET /views/other.html`. Então lembre-se **sistema de arquivos e sistema de rotas não é a mesma coisa**.

Quando utilizamos `src` ou `href` com um servidor HTTP, estamos navegando sobre rotas, e não arquivos. Perceba que, abrindo o documento `index.html` direto do navegador, o hyperlink funciona normalmente. Podemos consertar o problema de várias formas, vamos fazer da mais simples:

```
<!-- index.html -->
<a href="/views/other">...</a>
```

```
<!-- other.html -->
<a href="/">...</a>
```

```
...
app.get('/', function (request, response) {
  response.sendFile(path.join(__dirname, 'index.html'))
})

app.get('/views/other', function (request, response) {
  response.sendFile(path.join(__dirname, 'views/other.html'))
})
...
```

Agora associamos as rotas:

- `/ -> index.html`
- `/views/other -> other.html`

Com isso, a navegação funciona como esperado
