# **Cursos de JavaScript na Web – Alura**

## **Curso 1 – Manipule o DOM com JavaScript**

### **Aula 1 – Conhecendo o DOM:**

* 1. Tudo no DOM é um objeto e todo objeto possui queries. Através delas nós selecionamos as classes ou id das tags para podermos alterar o que quisermos através do JS. Para usar a query nós colocamos o objeto separado por “.”, A função seguida de parênteses e dentro dos parênteses colocamos aspas com o indicador da marcação ou a tag em si, e o nome dela dentro para determinarmos qual/quais elementos estamos pegando. Ex.: *document.querySelector(‘.classe’/’#id’/’tag’).*
     1. O query sempre irá devolver o primeiro elemento correspondente que encontrar na busca.
  2. Podemos também pegar somente o conteúdo de determinado objeto, utilizando a propriedade *textContent*, por exemplo: *document.querySelector(‘.content’).textContent*. Dessa forma ele irá exibir somente o conteúdo, ao invés da tag, sua classe e/ou id e depois o conteúdo.
     1. Utilizando esta técnica de seleção de elemento, também podemos alterar o conteúdo de uma tag utilizando a mesma linha acima: *document.querySelector(‘.content’).textContent = ‘comprar maça’*. Dessa forma, o texto anteriormente colocado na/s tag/s com essa classe será/ão alterados para “comprar maça”, independentemente do que estivesse escrito antes.
  3. Como a árvore de hierarquia do DOM funciona:



* 1. Outros métodos de seleção de elementos além do querySelector(‘’):
     1. *document.getElementById(‘id’)*: seleciona o elemento pelo id passado.
     2. *document.getElementsByClassName(‘classe’)*: retorna um array dos elementos pelo nome da classe passada.
     3. *document.getElementsByTagName(‘tag’)*: retorna um array dos elementos pelo nome da tag passada
     4. *document.querySelectorAll(seletor)*: devolve todos os seletores com o mesmo nome

1. **Aula 2 – Comportamento do Formulário:**
   1. O JS é sempre colocado por último no html devido a ordem de carregamento ser de cima para baixo, pois dessa forma, caso tenha algum problema no nosso código, evita que o carregamento da página seja interrompido como seria se ele tivesse sido colocado por último, mantendo o cliente dentro do nosso site.
   2. Podemos importar um arquivo JS colocando um *src=”localDoArquivo”* dentro da tag *<script>*. Não precisamos escrever o código juntamente com o html.
   3. Um formulário tem suas próprias propriedades e uma delas é recarregar a página quando o botão é pressionado, mesmo que não aconteça nada.
   4. Para fazer com que um item seja adicionado a uma lista após escrevê-lo no input text de um forms, precisamos mexer no botão pela árvore DOM usando JS, ao invés do input.
   5. Existe um quarto seletor que podemos utilizar para selecionar itens específicos por querySelector no JS chamado de: *“data-atributes”*. Basta colocar a palavra *“data-“* e o nome que deseja dar àquele elemento no html.
      1. Ele serve como um separador de responsabilidade.
      2. A principal vantagem é separar o que é do css e o que é do JS, pois se utilizarmos as classes outra pessoa que trabalha no mesmo projeto pode acabar mudando a classe por achar que não está semântica para o CSS e acabar quebrando o código. Utilizando os *data*’s esse risco é bem menor.
      3. Para se referir ao *data* colocado no html, no querySelector, precisamos colocar “[]” dentro das ‘’ e, dentro dele, o nome exato do *data* utilizado para se referenciar àquele elemento. Ex.: *document.querySelector(‘[data-nome-atribuído-ao-elemento]’)*.
   6. Podemos colocar qualquer expressão que será utilizada várias vezes dentro de uma variável constante.
      1. No caso do projeto referente à este primeiro curso, a variável é o button, então podemos colocar: *const novaTarefa =* *document.querySelector(‘[data-form-button]’)*.
      2. Desse modo, sempre que quisermos utilizar essa expressão toda dentro do nosso código, podemos colocar a variável que não muda “novaTarefa”. Isso é válido para qualquer expressão.
   7. Para fazer com que o botão faça uma ação quando for clicado precisamos adicionar um Evento de monitoramento à ele, para que sempre que for clicado, execute uma função específica. Traduzindo isso para o JS fica literalmente assim: *novaTarefa.addEventListener(‘click’, ()=>{console.log(‘fui clicado’)})*, respectivamente. Ou seja, vc adicionou um evento de ouvir para o botão que agora está com o nome de novaTarefa e disse para ele que quando ele “ouvir” que o botão foi clicado, irá executar a função anônima que é imprimir a frase “fui clicado” no console do browser.
      1. O problema é que ele irá aparecer e sumir muito rápido, para preservar o log do console você precisa clicar nas settings dele e marcar a opção “preserve log”. Dessa forma ele não desaparece.
   8. Podemos pegar somente o valor de um input, ou seja, seu texto, utilizando o *nomeDaVariável/expressãoDoInput.value*, dessa forma pegando apenas o que estiver escrito no input, ao invés de toda sua expressão html.
      1. Para mandar ele imprimir no console o texto do input, podemos colocar essa expressão dentro da função anônima do event listener, que ocorre quando apertamos o botão.
      2. Lembrando que podemos criar variáveis para todas essas expressões, facilitando o entendimento do código.
   9. Nós conseguimos fazer com que o formulário pare de atualizar a página, ou seja, pare de mandar informações para o servidor colocando um *preventDefault()*, dentro do eventListener.
      1. Primeiro precisamos nomear aquela função anônima que irá imprimir o texto do input no log e em seguida, dentro da função, colocamos o nome dela seguido de um ponto e a sentença acima.
      2. Fazendo isso, assim que clicarmos no botão o formulário não irá atualizar a página como estava fazendo antes.
   10. Para organizar melhor o nosso código, nós podemos criar uma função para não deixar tudo bagunçado no eventListener.
       1. Para criar uma função precisamos de uma constante com o nome da função recebendo o evento entre parênteses e todos os parâmetros que estavam dentro do eventListener.
       2. Agora podemos colocar apenas o nome da função depois da vírgula no eventListener ao invés de toda a função que estava previamente.
   11. Mais informações sobre *data-atributes*: <https://cursos.alura.com.br/data-attributes-do-html5-c109>
   12. O que aprendemos na aula de hoje:
       1. Utilizar data-attributes;
       2. Utilizar o método addEventListner para escutar eventos no elemento;
       3. Prevenir o comportamento padrão do formulário.
2. **Aula 3 – Adicionar Item na Lista:**
   1. Para adicionarmos conteúdo no html de forma dinâmica nós utilizamos o parâmetro *InnerHTML* seguido da variável que queremos alterar, recebendo o conteúdo alterado.
      1. Isso não adiciona um novo conteúdo, apenas substitui o anterior dependendo das variáveis e elementos selecionados.
   2. Para criarmos conteúdo/tags dentro do DOM/HTML, precisamos utilizar a propriedade *document.createElemet(‘tag’)*.
   3. Como existe hierarquia de parentesco nas tags HTML, precisamos colocar o parâmetro *tagMãe.appendChild(tagFilho)*. Ao fazer isso o novo conteúdo/tag será incorporado ao HTML dinamicamente.
      1. Porém, ao fazer isso o texto não vem com a formatação CSS, será apenas um texto incorporado ao corpo do HTML sem nenhuma estilização.
      2. Para contornar esse problema, adicionamos a classe do css que já existe para esse elemento utilizando o comando: *variávelDoElemento.classList.add(‘classeCSS’)*.
      3. Fazendo isso todos os novos elementos referentes a essa variável criados dinamicamente irão apresentar a classe escolhida, que já estará configurada no CSS.
      4. Depois disso tudo, o conteúdo adicionado dinamicamente estará totalmente estilizado da maneira correta.
      5. O *appendChild* sempre cria um novo elemento no final no do nó, portanto, sempre que um novo item for criado ele será alocado logo após o anterior.
   4. Vídeo da Alura explicando melhor sobre template strings: <https://cursos.alura.com.br/template-string-c123>
   5. Todos os elementos na nossa árvore do DOM são nós e todos os nós podem ser acessados via JavaScript. Os nós podem ser deletados, criados ou modificados. Durante o curso utilizamos o método *appendChild* que sempre é adicionado no final do nó, para colocar um nó filho dentro do nó mãe. Outros métodos para manipular nós:
      1. *insertBefore(pai, filho)*: Coloca um nó antes do outro;
      2. *replaceChild( elemento1, elemento2)*: Substitui o nó elemento 1 pelo nó elemento2;
      3. *removeChild(elemento)*: Remove um nó da árvore.
   6. O que aprendemos na aula:
      1. Utilizar template strings;
      2. Colocar um elemento filho dentro do elemento pai utilizando o método *appendChild*;
      3. Criar elementos utilizando o método *createElement*.
3. **Aula 4 – Concluir Tarefa:**
   1. Utilizando todos os métodos vistos até agora nós conseguimos criar e inserir botões em qualquer lugar de um HTML.
   2. Para alterarmos um estilo precisamos adicionar o comando *toggle* logo após a propriedade *classList* de um elemento. Dentro das ‘’ dentro dos () você coloca qual a propriedade CSS que deseja que aconteça.
   3. O *toggle* sempre devolve um booleano, ou seja, um valor verdadeiro ou falso.
   4. Um grande problema do JS é você deixar seu código no escopo global, pois dessa forma qualquer pessoa tem acesso ao seu código e suas funções de finidas.
      1. A solução para esse problema é criar uma função anônima externa ao código original, pois dessa forma, qualquer coisa que a pessoa digitar no console do browser aparecerá como indefinido.
      2. Para fazer isso basta colocar todo o código entre () e dentro de uma função anônima, ou seja: (() => { CódigoCompleto }). Dessa forma o nosso código fica protegido.
      3. Porém, ao tentar executar o código dessa forma veremos que ele não irá funcionar. Para corrigir isso colocamos () no final de tudo, ficando assim: (() => { CódigoCompleto })(). Desse modo e com essas alterações, o código funcionará normalmente e ficará protegido.
      4. Essa técnica se chama IIFE (*Immediately Invoked Function Expression ou Função de Invocação Imediata*).
   5. O que aprendemos na aula de hoje:
      1. Adicionar classe CSS utilizando o método toggle;
      2. Utilizar o atributo parentElement para subir um elemento na árvore do DOM;
      3. Encontrar o alvo do evento utilizando a propriedade target;
      4. Utilizar IIFE.
4. **Aula 5 – Remover Tarefa:**
   1. Sempre que criamos uma função de um componente, o nome começa com letra maiúscula.
   2. Para remover um objeto/elemento utilizamos a propriedade *.remove()* no JS.
   3. Módulos são pequenas partes do código que formam um todo.
   4. Para criar módulos precisamos exportar o código de um arquivo que é o módulo e importar nos arquivos que queremos utilizar esse módulo.
      1. Para exportar usamos: *export default nomeDaFunção/Elemento*.
      2. Para importar usamos: *import nomeDaFunção/Elemento from “./localização”*
      3. Para exportar mais de um objeto do módulo colocamos: *export{ objeto1, objeto2 }*
   5. Para dizer para o nosso código que ele é um módulo, precisamos ir no HTML onde o script está sendo importado e colocar uma propriedade *type=”module”* confirmando esse fato.
   6. O que aprendemos na aula de hoje:
      1. Utilizar import/export;
      2. Remover elementos do DOM com o método remove;
      3. Entender Same Origin Police e CORS.

**Curso 2 – Armazenando Dados no Navegador:**

1. **Aula 1 – Trabalhando Com Datas:**
   1. Criando um componente html chamado *input*, mas com o *type=”datetime-local”* já estamos criando um input de seleção de data, ou seja, algo que abra um calendário para selecionarmos uma data.
   2. Para padronizar as datas podemos utilizar a biblioteca: <https://momentjs.com>.
      1. No próprio site tem exemplos de como utilizar a biblioteca no seu código.
   3. Para baixar uma biblioteca podemos entrar no site <https://cdnjs.com> e procurar pela biblioteca que queremos, no caso a citada acima.
      1. Ao achar a biblioteca que deseja e entrar no link dela, basta copiar a primeira URL disponibilizada.
      2. No seu index.html tudo o que precisa fazer é criar um outro *<script>* e colar a URL no *src=””* dele.
   4. Depois de adicionar a biblioteca, no nosso código podemos passar o formato que desejamos que a data seja exibida como exemplificado no site do momentjs.
      1. Basta colocar como parâmetro na constante data criada anteriormente e que se encontra no *console.log(data.format(‘DD/MM/YYYY’))*, ficando dessa maneira.
   5. Podemos utilizar parâmetros como *.toLocaleDateString(‘pt-br’, dataOptions)* para datas e *toLocaleTimeString()* para horas, desta forma, fazendo com que a configuração/formato da data e hora seja exibida de acordo com os parâmetro utilizados na localização do acesso da aplicação.
   6. Podemos configurar a data e a hora de acordo com o que queremos criando uma constante da seguinte forma:
      1. const dataOptions = {
      2. weekend: 'long',
      3. year: 'numeric',
      4. month: 'long',
      5. day: 'numeric'
      6. }
      7. const horarioOptions = {
      8. hour12: false,
      9. hour: 'numeric',
      10. minute: '2-digit',
      11. second: '2-digit',
      12. timeZone: 'America/Sao\_Paulo'
      13. }
   7. Alura + explicando como e o que é a desestruturação no JS: <https://cursos.alura.com.br/destructuring-em-js-c308>
   8. O que aprendemos:
      1. Utilizar componente de calendário;
      2. Manipular datas;
      3. Desestruturar objetos em JavaScript.
2. **Aula 2 – LocalStorage:**
   1. Podemos utilizar a API do navegador para criar um código e armazenar informações nele.
   2. Site com a documentação para pesquisar e estudar sobre api: <https://developer.mozilla.org/pt-BR/docs/Web/API/Web_Storage_API>
   3. Para utilizar o armazenamento utilizamos o *sessionStorage* e colocamos o *setItem(“nome”, JSON)* como parâmetro para ele.
      1. Dessa forma damos um nome ao storage e colocamos os dados que desejamos que ele armazene.
      2. Se houver mais de um, podemos criar uma constante e colocar todos eles dentro dela, reduzindo a quantidade de informação para apenas “dados”, dessa forma facilitando o processo.
      3. Como esse método só pode ser usado se o segundo argumento for um JSON que é formato de string, precisamos fazer a conversão desses dados para dar certo.
      4. Para converter os dados utilizamos o *JSON.stringfy(dados)*.
      5. O sessionStorage faz o que diz, ou seja, armazena os dados enquanto aquela sessão estiver iniciada, a partir do momento que ela se encerra, os dados são apagados.
   4. Explicação do que é API (*Application Programming Interfaces*): <https://developer.mozilla.org/pt-BR/docs/Aprender/JavaScript/Client-side_web_APIs/Introdu%C3%A7%C3%A3o>
   5. Como o sessionStorage não serviu para a nossa aplicação, vamos utilizar o *localStorage* no lugar. Os parâmetros e argumentos são todos os mesmos, só muda o primeiro nome.
      1. Com o local os dados se mantém mesmo se fechar a aba e abir o mesmo site em outra.
      2. Lembrando que o que está escrito visualmente na aplicação será apagado, mas os dados estarão armazenados e podemos vê-los na parte de aplicação do console do navegador.
      3. Utilizando essa técnica, sempre que colocamos uma nova tarefa na nossa aplicação a que estava anteriormente será apagada e substituída pela nova.
      4. Para corrigir isso precisamos acrescentar código na nossa aplicação.
   6. Para manter os dados armazenados podemos criar uma Array vazia e chamar ela dentro da nossa função de adicionar novo item mandando fazer um *tarefas.push(dados)*, dessa forma, todos os dados inseridos nos inputs da tarefa serão jogados para dentro da array, mantendo os dados salvos ao invés de serem substituídos como ocorria anteriormente.
   7. Podemos pegar itens do localStorage usando o *.getIten(‘const para armazenar’)*.
      1. Para passar todos os dados que estão em formato de string de volta à objeto, utilizamos o *JSON.parse(localStorage.getIten(‘local’)) || []*.
      2. Os dois pipes || e [] no final indica que se não tiver nada lá, ele irá iniciar/tratar isso como uma Array.
   8. O que aprendemos:
      1. Salvar dados no navegador utilizando *SessionStorage*;
      2. Armazenar dados sem encerrar a sessão com e *LocalStorage*;
      3. Transformar objetos em string utilizando *JSON.stringify()*.
3. **Aula 3 – Renderizando a Lista:**
   1. LocalStorage se comporta como um arrayLike, que nos permite utilizar métodos de Array dentro dele.
   2. O que aprendemos:
      1. Iterar sobre os itens do *LocalStorage*;
      2. Utilizar o curto circuito para verificação lógica;
      3. Converter os itens que antes eram strings em objetos com *JSON.parse*.
4. **Aula 4 – Ordenando as Tarefas:**
   1. O que aprendemos:
      1. Ordenar elementos do *LocalStorage*;
      2. Filtrar elementos de um Array.
5. **Aula 5 – Finalizando o Projeto:**
   1. O que aprendemos:
      1. Alterar um estado utilizando operador de negação;
      2. Remover dados do *LocalStorage* utilizando o método *splice*.

**Curso 3 – Expressões Regulares – Capturando Textos de Forma Mágica**

1. **Aula 1 – Começando com Regex:**
   1. Regex = Expressões regulares.
   2. Podemos filtrar os textos usando as expressões.
   3. Existem alguns caracteres que possuem um significado especial para o regex engine. Especial significa que o regex engine não interpreta o valor literal e sim diferente. Esses caracteres são chamados de meta caracteres.
      1. . o "ponto" que significa qualquer char
      2. \* o asterisco que serve para definir uma quantidade de chars, zero ou mais vezes
      3. {e } as chaves que servem para definir uma quantidade de caracteres específicas que é desejado encontrar
      4. Exemplos:
         1. a{3} letra a 3 vezes.
         2. \d\* um digito zero ou mais vezes
      5. Lembrando também, se quisermos procurar pelo \* ou . literalmente (sem significado especial), devemos utilizar o caractere \
   4. CSV: valores separados por vírgula (coma separeted values).
   5. Colocando */d* (dígito) no pesquisador ele irá exibir todos os resultados que possuem números.
   6. Quantifier: Conjunto de caracteres especiais que definem quantas vezes um caracter deve aparecer. Utilizamos {n} para definir isso.
2. **Aula 2 – Classes de Caracteres:**
   1. ?: Caracter especial para dizer que aquela informação pode existir uma ou nenhuma vez na string que estamos procurando. Também podemos dizer isso utilizando o {0,1}, mas temos uma caracter específico para isso.
   2. Podemos definir um conjunto/classes de caracteres e dizer que eles podem ou não aparecer na string utilizando: []. Tudo o que estiver dentro desses colchetes já estão no seu significado literal, ou seja, não precisamos mais colocar o \. Para dizer que queremos literalmente o “.” E não seu significado na língua. O próprio \d é uma classe de caracteres abreviada, é a mesma coisa que dizer [0123456789].
   3. - : utilizado para dizer de um até outro em uma classe: [1-36-9] = 1,2,3 e 6,7,8,9.
   4. \s: White spaces, ou seja, ele irá contar com o espaço em branco subsequênte, e se colocarmos {1,}, não importa se foi dado um space ou tab, pois estamos especificando que pode ser 1 ou infinitos espaços. Porém, existe uma forma mais simples sendo ela “+” e significando a mesma coisa.
   5. [^,]: classe para determinar que pegaremos tudo o que estiver antes da vírgula, não importando se é letra, número, caracter especial ou a quantidade que tenha.
   6. Somente \^- ainda são considerados meta-chars em classes.
   7. ^: Significa negação dentro dos colchetes.
   8. O que aprendemos:
      1. Podemos definir facilmente a classe de qualquer caractere com o [A-Z].
      2. Conhecemos todos os quantifiers como ?, +, \* e {n}.
      3. \s significa whitespace e é um atalho para [ \t\r\n\f].
      4. \w significa word char e é uma atalho para [A-Za-z0-9\_].
3. **Aula 3 – Encontrando Posições Certas Com Âncoras:**
   1. \w: wordchar, ou seja: [A-Za-z0-9\_]
   2. Âncoras te devolvem a posição do texto e não ele em si.
   3. \b: word boudary, ou seja, essa ancora usada antes e/ou depois de uma string diz para o software que não queremos que ele selecione nenhuma palavra que qualquer \w além dos desejados.
   4. Existem várias âncoras predefinidas, mas as mais comuns são ^, $ e \b. Lembrando também que os caracteres ^ e $ são meta-chars.
   5. Uma âncora não casa caracteres como as classes fazem, e nem definem quantidades. Âncoras marcam uma posição específica no alvo, por isso não é possível combiná-las com um quantifier.
   6. ^: significa “início”, fora de colchetes.
   7. $: significa “final”.
   8. . : significa qualquer caracter, ou seja, não importa o que estiver lá.
   9. \B: non-word-boundary, basicamente diz que a sílaba/palavra utilizada não pode estar no início e nem no fim de uma str, sempre no meio dela. É o oposto da \b.
   10. O que aprendemos:
       1. Existem âncoras predefinidas que selecionam uma posição dentro do alvo.
       2. \b é uma âncora que seleciona um word boundary, isso é o início ou fim da palavra.
       3. ^ é uma âncora que seleciona o início da string alvo.
       4. $ é uma âncora que seleciona o fim do alvo.
4. **Aula 4 – Trabalhando com Grupos:**
   1. Podemos separar as seleções por grupos colocando () na parte regex que queremos dar ênfase.
   2. Colocando (?:regex) faz com que esse grupo não seja capturado no resultado final do regex engine. O nome dessa técnica é *non-capturing-groups*.
   3. <http://www.goulart.pro.br/cbasico/Calculo_dv.htm> Mais sobre o cálculo 11 de CPF’s.
   4. Mini-calculador para usar o Cálculo 11
   5. [http://www.cjdinfo.com.br/utilitario-calculo-digito-modulo-11](http://www.cjdinfo.com.br/utilitario-calculo-digito-modulo-11 )
   6. | : indica condição OU em um grupo para poder selecionar e-mails, por exemplo.
   7. O que aprendemos:
      1. Declaramos um grupo com ().
      2. Podemos ter grupos e subgrupos.
      3. Um grupo é retornado na hora de executar, são úteis para selecionar uma parte do match.
      4. Através do ?:, dizemos que não queremos ver esse grupo na resposta
5. **Aula 5 – Ganancioso ou preguiçoso:**
   1. ? : quando depois do + faz com que ele deixe de ser ganancioso, ou seja, pegue o máximo de caracteres possível e passe a ser preguiçoso, ou seja, pare no primeiro caracter especificado depois dele.
   2. Sempre criar um grupo para fazer comparações.
   3. \numero: Referência à um grupo na sua regex, já que eles são numerados de 1 ao infinito, desse modo podemos fazer condições falando para o regex que se ele selecionou uma das opções do grupo 1, por exemplo, aquela opção vai ter que ser a mesma, do contrário não seleciona a str. O nome é backreferences.
   4. O \W é a non-word char, ou seja tudo que não é um word char. \W é um atalho para [^\w]
   5. A classe \D, por sua vez, é um non-digit, ou seja, \D é um atalho para [^\d]
   6. O que aprendemos:
      1. Aprendemos que quantifiers são gananciosos por padrão e que podemos utilizar um ? logo após o quantifier, deixando-o preguiçoso. Também aprendemos como podemos referenciar o texto de um grupo dentro da regex, aonde n é o número do grupo.
6. **Aula 6 – Usando Regex nas Diversas Linguagens:**
   1. 
   2. 
   3. 
   4.  
   5.  
   6. 
   7. 
   8. 
   9. 
   10. 
   11. 
   12. 

**Curso 4 – CRUD com JS Assíncrono:**

1. **Aula 1 – Comunicação assíncrona:**
   1. New XMLHttpRequest(): requisita comunicação com o servidor. Pode ser colocado dentro de uma variável constante para facilitar, ex: http.
   2. http.open(‘GET’, ‘http://localhost:3000/profile’): Obtem dados a partir do caminho dado dentro da comunicação requisitada.
   3. http.send(): Envia os dados de volta.
   4. http.onload = () => {

const data = http.response

console.log(data)

}: Ao carregar a variável data recebe a resposta do http e imprime no console o que recebeu.

* + 1. Podemos substituir o console por:
    2. tabela.appendChild(criaNovaLinha(elemento.nome,elemento.email))
    3. dessa forma criando uma nova linha (através de uma função já feita antes com o creatElement, etc...) com as informações/dados que queremos que sejam exibidas no site.
    4. Fazendo isso, ao invés de exibir no console, os dados aparecem no site.
  1. Função criada para criar a linha:
     1. const criaNovaLinha = (nome, email) => {
     2. const linhaNovoCliente = document.createElement('tr')
     3. const conteudo = `
     4. <td class="td" data-td>${nome}</td>
     5. <td>${email}</td>
     6. <td>
     7. <ul class="tabela\_\_botoes-controle">
     8. <li><a href="../telas/edita\_cliente.html" class="botao-simples botao-simples--editar">Editar</a></li>
     9. <li><button class="botao-simples botao-simples--excluir" type="button">Excluir</button></li>
     10. </ul>
     11. </td>`
     12. linhaNovoCliente.innerHTML = conteudo
     13. return linhaNovoCliente
     14. }
  2. O que aprendemos:
     1. Fazer requisições utilizando xmlhttprequest.
     2. Lidar com promises.
     3. Fazer uma requisição http utilizando xmlhttprequest para buscar todos os clientes do servidor.
     4. Refatorar o código utilizando promises melhorando a leitura do código.
     5. Utilizar template literals para criar um template html.

1. **Aula 2 – Fetch API:**
   1. A fetch já substituí tanto o xmlh quanto o promise, desse modo, podemos substituir várias linhas de código por:

return fetch(`http://localhost:3000/profile`)

* 1. E colocar um:

.then(*Responsta* => {

    return *Responsta*.json()

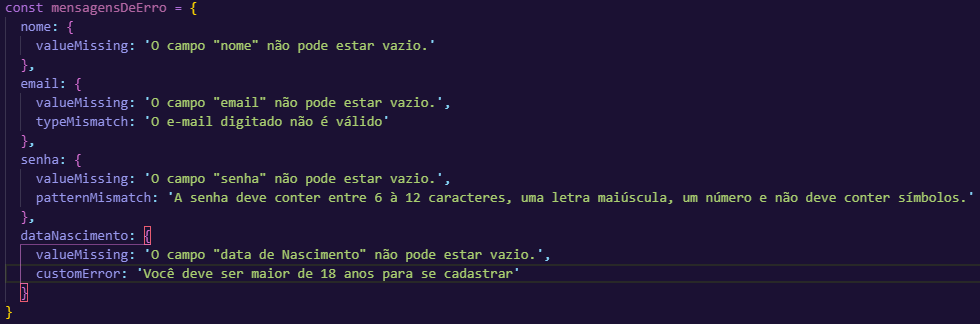
  })

* 1. Para devolver a resposta obtida.
  2. Artigo para aprender mais sobre fecth api:
  3. [https://www.alura.com.br/artigos/comecando-com-fetch-no-javascript](https://www.alura.com.br/artigos/comecando-com-fetch-no-javascript )
  4. JS só consegue executar uma coisa por vez devido sua época de criação e objetivo.
  5. Todas as funções decorrentes do WebAPI é executada depois das nativas do JS, mesmo que venham antes na ordem da escrita.
  6. Site para simular o ambiente de funcionamento do JS: <http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7ICAgIAogICAgfSwgMjAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5sb2coIldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D>
  7. Site da globo de 2006 para ver a funcionalidade do JS sem o uso do WebAPI: <https://web.archive.org/web/20060516060134/http://globoesporte.globo.com/>
  8. O que aprendemos na aula:
     1. Entender a ordem de execução do código JavaScript.
     2. Puxar dados do servidor utilizando a função a fetch api ao invés do xmlhttprequest para modernizar o código.
     3. Refatorar as responsabilidades do código pensando na manutenção da aplicação no futuro.

1. **Aula 3 – Criar e Remover Clientes:**
   1. Para postar algo no servidor usamos o method: ‘POST’ dentro do return fetch(`http://localhost:3000/profile`,{aqui}).
   2. Ao invés de exportar cada função separadamente e depois importar no arquivo de execução, podemos criar uma constante com as funções que queremos exportar dentro dela e mandar exportá-la.
   3. Sempre que quiser pegar o valor dentro de uma caixa de texto em um formulário coque *.value* no final do querySelector.
   4. No final do http podemos colocar ${id} para indicar que o que acontecerá naquela função é específica para um cliente e não é para acontecer com todos.
      1. Colocar id na function também (id) => {...}
   5. Para deletar basta passar o method: ‘DELETE’
      1. Não esquecer de colocar dentro do fetch(http:..., {aqui})
   6. Podemos criar um data-atributes usando o *dataset.tipoDeAtributo*.
   7. Para saber mais sobre comunicação entre cliente e servidor: <https://cursos.alura.com.br/o-que-e-rest--c119>
   8. O que aprendemos:
      1. Usar a método closest para encontrar o elemento do DOM mais próximo ao que queremos remover;
      2. Criar um cliente utilizando o verbo http POST;
      3. Remover um elemento do dom com método remove();
      4. Deletar um cliente utilizando o verbo http DELETE.
2. **Aula 4 – Editar Dados:**
   1. Window.location = pega a localização da página em que está no site.
   2. O que aprendemos:
      1. Fazer uma query string utilizando a propriedade serachParams.get() para encontrar um id;
      2. Criar uma IIFE ou função auto executável;
      3. Editar os dados utilizando o verbo http PUT.
3. **Aula 5 – Async/Await:**
   1. Async()=>{}: indica que essa é uma função assíncrona.
   2. Await: substitui o .then (()=>{}) quando o acima está .
   3. Site sobre as funções assíncronas: <https://developer.mozilla.org/en-US/>
   4. throw new Error(‘mensagem’): Lança um erro no console. Pode colocar mensagem personalizada.
   5. Try{código}: tenta executar o código dentro, caso não consiga:
   6. Catch{código}: exibe uma mensagem de erro ou executa outro código.
   7. O que aprendemos:
      1. Utilizar async/await para indicar funções assíncronas;
      2. Tratar erros com try/catch.

**Curso 5 – Validação de Formulários e HTML5**

1. **Aula 1 – Validação com HTML:**
   1. Adicionar o atributo *required* no HTML de um input de um formulário faz o preenchimento dele se tornar obrigatório
   2. Colocar o *type=”email”* no lugar do *type=”text”* em um input, faz o HTML exigir que a estrutura do texto colocado seja igual a de um e-mail, mas sem verificar se ele realmente existe.
   3. Colocar o *type=”password”* faz o que é digitado no campo ser substituído por \*\*\*.
   4. *minlength="n"*: Defini uma quantidade mínima de caracteres que precisa ter no campo.
   5. *Pattern=”regex”*: define um padrão necessário de senha a partir de uma regex para definir o que a senha deve ou não ter, como letra maiúscula, símbolo ou não, número ou não, letras minúsculas e quantidade de letras.
   6. *Title=”msg”*: Atributo cuja mensagem dentro será exibida junto com a mensagem de erro caso tente colocar uma senha fora do pattern, por exemplo.
   7. Ferramenta para treinar regex: <https://regexr.com/>
   8. O que aprendemos:
      1. Fazer validações no próprio HTML utilizando os atributos required e type;
      2. Usar regras de regex dentro do atributo pattern para validar o campo de senha;
      3. Mandar uma mensagem customizada de erro no balão do navegador.
2. **Aula 2 – Validações Customizadas:**
   1. *Type=”date”*: Faz o HTML verificar se o texto no input é uma data ou não.
      1. Ele cria uma máscara automaticamente não permitindo que nada além de uma data seja passada para aquele campo.
   2. *Function nomaDaFunção(){}*: Cria funções.
   3. *setCustomValidity(‘mensagem’)*: propriedade para setar uma mensagem de validação customizada em JS.
   4. *Data.getUTCFullYear()*: pega o ano completo atual.
   5. *Data.getUTCMonth()*: pega o mês atual.
   6. *Data.getUTCDate()*: pega o dia atual e joga tudo na variável data.
   7. *New Date()*: Pega a data atual.
   8. *If (!condição){código}*: o ! indica negação, ou seja, se a condição seguinte for falsa, execute o código.
   9. *addEventListener(‘blur’, função)*: O blur significa que a função será executada quando o campo sair de foco.
   10. O que aprendemos:
       1. Criar funções para validações customizadas do formulário;
       2. Definir mensagens customizadas para validações fora do HTML;
       3. Utilizar data attributes para trabalhar com JavaScript;
       4. Deixar a função de validação mais genérica para funcionar com qualquer input.
3. **Aula 3 – Mensagens Customizadas:**
   1. *Type=”module”*: Colocar junto com o <script src=”caminho”> para indicar que está usando o JS com módulos, caso contrário, ocorrerão erros de execução.
   2. Live server: Extensão para o VScode para deixar seu site/aplicação sempre rodando em um servidor no browser. Ao invés de abrir o html direto no browser, abra ele no VScode e clique em “go live” no canto inferior direito para ativar a extensão, ela irá abrir uma nova página no browser com seu site. Também serve para evitar erros.
   3. Browser-sync: Uma opção para quem usa outro editor de texto e que serve para a mesma coisa.
      1. Npm install -g (para ser global) browser-sync: Isso irá instalar o browser-sync.
         1. Utilizar o “sudo” antes do comando caso tenha dado erro. Não muito recomendado.
      2. Browser-sync start -s (abrir servidor) -f (checa arquivos do projeto) . (verifica todos os arquivos) –directory (abre a raiz do projeto no navegador): inicia o treco.
      3. OBS.: Ambas as extensões necessitam do node.js instalados no pc. A segunda opção precisa do npm instalado também.
   4. .*parentElement*: seleciona o elemento mãe do qual essa propriedade foi atribuída.
   5. *.classList*: se refere a classe de um elemento, podendo ser *.add(‘classe’)* ou *.remove(‘classe’)* daquele elemento referente.
   6. Podemos criar uma variável com vários objetos com mensagens de erros personalizadas para cada tipo de erro que aquele campo oferece. Podemos ver quais são os tipos de campo digitando *$0.validity* no console do navegador após selecionar o campo de input desejado, lá serão exibidos todos os tipos de erros existentes.
      1. Exemplo:



* 1. O que aprendemos:
     1. A instalar e usar o plugin Liveserver;
     2. Como baixar e rodar o browser-sync;
     3. Como mostrar a mensagem de erro diretamente no HTML;
     4. Como customizar as mensagens de erro de validação.

1. **Aula 4 – Validando CPF:**
   1. Podemos criar funções com vetores para fazer validação de números repetidos em cpf.
   2. Utilizando o *replace* com uma regex do tipo /\D/g fazemos a formatação do CPF inserido no input retirando tudo o que não for número.
   3. A conta para validação de CPF é:
      1. A soma de todos os dígitos multiplicados, por 10-2 ex:
         1. Soma = (10 \* 1)+(9\*2)+(8\*3)+...(2\*9)
      2. Seguido da divisão dessa soma por 11
      3. Seguido da subtração por 11
      4. A fórmula da conta fica assim: 11 – (soma / 11).
      5. Essa validação será subsequente para todos os dígitos do CPF, mas a próxima sempre dependerá da anterior, ou seja, se o primeiro número for válido, aí ele faz a verificação do segundo.
   4. Substr(n1,n2): Função do JS para cotação de string.
   5. O que aprendemos:
      1. A validar cpf com números repetidos;
      2. A matemática por trás da validação de um CPF;
      3. Recursão de função.
2. **Aula 5 – Conectando com a API ViaCEP:**
   1. Podemos usar ReGex para validar formatação de CEP’s.
   2. Fetch(url, options): serve para pegar a API de um site previamente setado na constante url seguindo as options que foram previamente criadas.
   3. Options{}: existem várias opções para setar, sendo algumas delas:
      1. Method: o que você quer que aconteça, geralmente usado com o *‘GET’* para pegar algo. É o tipo de requisição que será feita.
      2. Mode: geralmente usado com o *‘cors’*. Indica que a comunicação será feita entre aplicações diferentes.
      3. Headers{}: geralmente coloca-se 'content-type': 'application/json;charset=utf-8'. Diz como que queremos receber as informações da API.
      4. Essas são informações básicas para fazer uma requisição. Vale pensar da seguinte maneira: como queremos (method) e o que queremos (headers). A opção mode é opcional, mas ela é necessária quando vamos fazer chamadas entre aplicações diferentes.
   4. Para preencher os campos, basta fazer o querySelector de todos os campos e fazer o valor deles receber o valor correspondente do data obtido pelo fetc. Ex.:
      1. *const logradouro = document.querySelector('[data-tipo="logradouro"]')*
      2. *logradouro.value = data.logradouro*.
   5. O que aprendemos:
      1. Outra estrutura de regex para a pré validação do CEP
      2. Como fazer uma requisição para a API da ViaCEP
      3. Preencher outros campos do formulário com a resposta da API da ViaCEP
3. **Aula 6 – Aplicação de Máscaras:**
   1. A vantagem é fazer uma formatação prévia e a pessoa já coloca o valor de maneira esperada.
   2. Existe uma máscara já pronta no github: <https://github.com/codermarcos/simple-mask-money>
   3. Basta copiar e colar o script importando ele no html abaixo do seu script.
      1. Fazer um *if* e colocar o data-atributes do input de preço.
      2. Para usar basta coloca *SimpleMaskMoney.setMask(input, {config disponibilizada no site}*.
      3. Configure do jeito que for no país em que estiver.
   4. O que aprendemos:
      1. Como adicionar máscara monetária para o campo de preço.

**Alura+:**

1. **Aula 1 – Strict Mode:**
   1. Quando não declaramos uma variável, acabamos criando um objeto global, uma propriedade pertencente ao *window* que irá guardar a informação colocada, e não apenas uma variável.
   2. O modo estrito nos previne disso acontecer.
   3. Para ativar o modo estrito precisamos colocar: “use strict”. No começo do arquivo.
      1. Fazendo isso, nós recebemos um erro quando tentamos executar o código, caso tenhamos esquecido de declarar a variável.
      2. Desse modo ficamos protegidos de colocar objetos no escopo global.
   4. Colocar o modo estrito fará o código dar mais erros do que antes, mas é muito melhor aparecer os erros e concertar do que ter um código inconsistente rodando.
   5. Quando tentamos alterar um valor que não deveria ser alterado, normalmente ele só não se altera no código, mas não temos a informação de que deu errado, mas com o “use strict” ele nos diz que deu errado e qual foi o erro.
   6. Ele avisa que não é permitido deletar coisas que não podem ser deletadas, ao invés de só não acontecer e você não saber o porquê.
   7. Quando temos argumentos repetidos em funções ou código no geral, o primeiro é sempre substituído pelo segundo e assim por diante até chegar no último argumento repetido.
      1. Se tentarmos fazer uma função de soma usando como argumento a, a, c = 1, 3, 2 ; o JS substituirá o primeiro a = 1, pelo segundo a = 3.
      2. O retorno esperado seria 6, mas como ocorre essa substituição, o resultado final é 8.
      3. Isso ocorrerá sem nenhum aviso prévio ou que tem argumentos repetimos, mas se utilizarmos o modo estrito teremos essa informação e poderemos corrigir.
   8. O seu código nem sempre poderá ficar inteiro no modo estrito, mas podemos colocar ele em funções, classes, dentre outros para verificar se está tudo certo em determinada parte do código.